The Translation of High-Level Computer Languages to Other High-Level Languages

by

Raydon Arthur
R. A. Freak

B. Sc., Dip. Ed. (New England), Dip. Comp. Sc. (Newcastle)

submitted in fulfilment

of the requirements for the degree of

Master of Science

UNIVERSITY OF TASMANIA

HOBART

August 1979

Errata

p8, lines 1 & 2
replace "adaption" by "adaptation"

p22, 4th line from bottom
after "variable" insert "sized"

p39, line 2
replace "an integer to a negative" by "a negative number to a non-integer"

p60, 2nd line from bottom
before "." insert "(the RETURN is implied when the last end is reached)"

p80, line 3
after "." insert "For clarity, the form involving booleans is used when there are two variant parts"

p109, 2nd line from bottom
add "If the first item after the $ is not SET, RESET or POP then all options listed are SET and all others are RESET. This conforms to Burroughs Algol."

p152, line 5
add "Correctness was established by executing a Pascal program produced and checking the results, or visually comparing the FORTRAN and Pascal listings when procedures were translated"

p153, 9th line from bottom
replace "FORTRAN subroutines" by "Pascal procedures"

p154, line 9
replace "do" by "would"

p154, line 10
replace "saves" by "would save"

p154, line 16
replace "submitting" by "to submit"

p163, line 4
delete "workable"

p168, line 3
replace ". The" by "for special kinds of programs and subroutine libraries in which"

p168, line 4
replace "and these limitations are far" by ". Here the limitations may be"

p168, line 8
replace "low-level" by "lower level high-level"

p169, after "." add

"Further work should be done on the translator before it could be considered suitable for use in a production environment. Many of the 'standard' extensions incorporated in most versions of FORTRAN IV should be included in the translator so that working FORTRAN subprograms can be submitted to the translator without first being converted to standard FORTRAN. Such extensions would include the extensions to DATA statements, allowing DO statements to use expressions as the parameters, allowing identifiers longer than 6 characters, ... etc. The possibility of rewriting the translator in Pascal should be investigated, as the translator then would have more impact on the user community. However, the translator uses random access I/O on its intermediate file and this problem would have to be addressed before an attempt was made to rewrite the translator in Pascal."

Except as stated herein, this thesis contains no material which has been accepted for the award of any other degree or diploma in any university. To the best of my knowledge and belief the thesis contains no copy or paraphrase of material previously published or written by another person, except where due reference has been made in the text of the thesis.

R. A. Freak

CONTENTS

0. ABSTRACT vii

1. INTRODUCTION 1

2. HISTORY AND BACKGROUND INFORMATION 3

2.1 History of FORTRAN 3

2.2 History of Pascal 4

2.3 Structured Programming History and Background Information 6

2.3.1 History 6

2.3.2 The Translator's Interpretation of Structured Programming 9

2.3.3 Theory and Techniques of Structured Programming 10

2.4 FORTRAN Conversions 12

3. FORTRAN TO PASCAL CONVERSION 14

3.1 Reasons for Translating FORTRAN to Pascal 14

3.2 Limitations on the Conversion of FORTRAN to Pascal 17

3.2.1 Input/Output 17

3.2.2 Complex and Double Precision Types 21

3.2.3 Variable Arrays 22

4. CONVERSION DETAILS 23

4.0 General Comments 23

4.1 Program Form 24

4.1.1 Character Set 24

4.1.2 Lines 24

4.1.3 Comment Lines 24

4.1.4 Continuation Lines 26

4.1.5 Statement Label 26

4.1.6 Identifiers 28

4.2 Data Types 30

4.2.1 Integer, Real and Boolean Types 30

4.2.2 Double Precision Type 30

4.2.3 Complex Type 31

4.2.4 Hollerith Type 32

4.3 Data and Procedure Identification 33

4.3.1 Constants 33

4.3.1.1 Integer Constant 33

4.3.1.2 Real Constant 33

4.3.1.3 Double Precision Constant 34

4.3.1.4 Complex Constant 35

4.3.1.5 Logical Constant 35

4.3.1.6 Hollerith Constant 35

4.3.2 Variables 36

4.3.3 Arrays 36

4.3.4 Procedures 37

4.3.5 Type Rules 37

4.4 Expressions 38

4.4.1 Arithmetic Expressions 38

4.4.2 Relational Expressions 41

4.4.3 Logical Expressions 42

4.4.4 Evaluation of Expressions 42

4.5 Statements 45

4.5.1 Executable Statements 45

4.5.1.1 Assignment Statement 45

4.5.1.2 GO TO Assignment Statement 48

4.5.1.3 Unconditional GO TO Statement 50

4.5.1.4 Assigned GO TO Statement 51

4.5.1.5 Computed GO TO Statement 54

4.5.1.6 Arithmetic IF Statement 55

4.5.1.7 Logical IF Statement 58

4.5.1.8 CALL Statement 59

4.5.1.9 RETURN Statement 60

4.5.1.10 CONTINUE Statement 60

4.5.1.11 STOP Statement 61

4.5.1.12 PAUSE Statement 62

4.5.1.13 DO Statement 63

4.5.1.14 Input and Output 67

4.5.1.14.1 READ and WRITE Statements 68

4.5.1.14.1.1 Input/Output Lists 69

4.5.1.14.1.2 Formatted READ Statement 71

4.5.1.14.1.3 Formatted WRITE Statement 71

4.5.1.14.1.4 Unformatted READ Statement 72

4.5.1.14.1.5 Unformatted WRITE Statement 72

4.5.1.14.2 Auxiliary Input/Output Statements 73

4.5.1.14.2.1 REWIND Statement 73

4.5.1.14.2.2 BACKSPACE Statement 74

4.5.1.14.2.3 ENDFILE Statement 74

4.5.1.14.3 Printing of Formatted Records 75

4.5.2 Non-Executable Statements 75

4.5.2.1 Array Declarator 75

4.5.2.2 DIMENSION Statement 77

4.5.2.3 COMMON Statement 77

4.5.2.4 EQUIVALENCE Statement 84

4.5.2.5 EXTERNAL Statement 87

4.5.2.6 Type Statements 87

4.5.2.7 DATA Statement 89

4.5.2.8 FORMAT Statement 90

4.6 Procedures and Subprograms 94

4.6.1 Statement Functions 94

4.6.2 Intrinsic Functions 96

4.6.3 External Functions 101

4.6.4 SUBROUTINE subprograms 106

4.6.5 BLOCK DATA subprograms 107

4.7 Non-Standard Features 108

4.7.1 IMPLICIT Statement 108

4.8 Translator Options 108

4.8.1 B6700/CDC/ICL/STANDARD 112

4.8.2 COMPLEX 112

4.8.3 DISK 113

4.8.4 DOUBLE 113

4.8.5 FLIST/PLIST 113

4.8.6 FORSTMT 114

4.8.7 IDLENGTH 114

5. SPECIFIC IMPROVEMENTS 115

5.1 Structure 115

5.1.1 Techniques of Structuring Programs 116

5.1.2 Structuring used by the Translator 126

5.2 Subprogram Ordering 130

5.3 Pascal Program Layout 132

6. FIELD TRIALS 136

6.1 Specific Test Cases 136

6.1.1 Lattice Structure with Abnormal Selection Paths 136

6.1.2 Loop with a Single Entry and Multiple Exits 138

6.1.3 Simple Subroutine Linkage with COMMON Blocks 143

6.1.4 Recursive Subroutine Linkage 147

6.2 Real Test Cases 152

7. EVALUATION OF RESULTS 163

8. CONCLUSION 168

APPENDIX

A. THE EFFECT OF THE NEW FORTRAN STANDARD ON THE TRANSLATOR 170

B. LISTINGS and EXAMPLES (Microfiche) back cover

B.1 Translator Listing

B.2 Translator Cross-Reference Listing

B.3 A Selection of FORTRAN to Pascal Translations

BIBLIOGRAPHY 186

0. ABSTRACT

This thesis is an investigation into the extent to which the language, FORTRAN, can be converted automatically to Pascal. Much has been written and said about the advantages of a "structured" language but there seems to be some reluctance for the computing community, as a whole, to adopt a structured language. A large amount of investigative work has been carried out by other workers into the possibility of extending FORTRAN to include some of the concepts of structured programming and other people have begun projects to convert FORTRAN to other structured languages but this is the first known attempt to convert FORTRAN to Pascal.

The process adopted is not simply one of literally replacing FORTRAN by Pascal. Rather, an attempt is made to introduce the structured concepts of Pascal to the FORTRAN program. This involves replacing FORTRAN loops by the Pascal statements repeat ... until or while ... do, using the powerful Pascal if statement, and laying out the COMMON and EQUIVALENCE statements so that the structure is apparent. A clear, consistent layout procedure is adopted and the program structure in the Pascal listing produced is highlighted. FORTRAN subprograms are nested within the procedure which calls them, or declared global to the set of procedures which call them - an approach which eliminates the global subprogram declarations of FORTRAN.

1. INTRODUCTION

The work reported in this thesis was carried out during the years 1977-1979 while the author was a member of the staff of the Department of Information Science, University of Tasmania.

The project was first mooted in 1976 when the author, in conjunction with Professor A.H.J. Sale, was developing a Pascal compiler for the Burroughs B6700/7700 range of computers. We felt, at that time, that if Pascal were to become a popular language and replace FORTRAN, then some aids would have to be provided to assist in the conversion of FORTRAN programs, particularly large FORTRAN libraries, to equivalent Pascal programs.

The aim of the project was, therefore, to investigate the extent to which correct, standard FORTRAN programs could be converted to correct "standard" Pascal programs. It was felt that, rather than performing a simple transliteration, some of the properties of the language Pascal, particularly structure and procedure order, should be reflected in the programs produced. Where it is not possible to produce a correct standard Pascal program, either an informative message is produced or an assumption concerning the version of Pascal to be produced is made and the translation allowed to proceed under that assumption.

The tests and field trials carried out show that the project has been successful and indicate that this sort of language translaton is viable and an aid to language conversion.

This document has been prepared using the RUNOFF text editing

system and printed on a DIABLO 1620 terminal.

The author acknowledges the helpful suggestions and constructive criticisms of his supervisor - Professor A. H. J. Sale of the Department of Information Science, University of Tasmania. He is also indebted to his wife for her help in checking the manuscript.

2. HISTORY AND BACKGROUND INFORMATION

2.1 History of FORTRAN

The FORTRAN project was initiated by IBM in 1954 for the IBM 704 computer. The original objective in the first FORTRAN programming language was defined as:-

> "The FORTRAN language is intended to be capable of expressing any problem of numerical computation. In particular, it deals easily with problems containing large sets of formulae and many variables, and it permits a variable to have up to three independent subscripts. However, for problems in which machine words have a logical rather than a numerical meaning, it is less satisfactory, and it may fail entirely to express some such problems. Nevertheless, many logical operations not directly expressable in the FORTRAN language can be obtained by making use of provisions for incorporating library routines."

The first FORTRAN processor was modified in 1958 to accept programs written in an augmented FORTRAN lanugage, commonly known as "FORTRAN II". The usage of FORTRAN II grew rapidly and processors became available for a wide variety of computers of quite varied structure and power. The major vendors recognized the requirement to provide FORTRAN compilers in order to compete with IBM. Their general strategy was to provide a compiler with the functionability of the 704 FORTRAN but with additional capabilities.

Beginning in 1962, processors for "FORTRAN IV" began to appear and came into increasing use although FORTRAN II processors remained

in quite substantial use. Efforts to standardise FORTRAN date back to early 1960 when the language had just been selected by the computing industry over Algol as the de facto standard for scientific and engineering work. Standardisation committees were established to investigate FORTRAN II and FORTRAN IV and eventually a final standard was approved in March 1966, based on FORTRAN IV.

Based on standard FORTRAN, vast libraries of subroutines began to be developed and to come into widespread use. These libraries typically performed scientific, mathematical or statistical functions. The relative ease with which standard FORTRAN is portable led to these libraries being transferred to other manufacturers' machines and helped to spread the use and acceptance of standard FORTRAN.

By 1968, sufficient extensions had appeared in actual implementations for an attempt to be made to standardise them. The standardisation bodies decided, however, to conduct a complete revision rather than a review. The revision took a number of years before a proposed standard was published in 1976 [Fortran 1976]. The work involved several hundred technical proposals from all over the world and is estimated to have cost in excess of two million dollars. The standard was available for comment for some time and during that time a number of alterations were made to it. The standard was approved in April 1978.

2.2 History of Pascal

A preliminary version of the programming language Pascal was drafted in 1968 by Niklaus Wirth. It followed, in its spirit, the

Algol-60 line of languages. After an extensive development phase, the first compiler became operational in 1970, and Wirth published his results in 1971 [Wirth 1971 a and b].

At that time, interest was growing in the development of compilers for other computers, and this called for a consolidation of Pascal. Also, two years of experience in the use of the language dictated a few revisions. This led in 1973 to the publication of a Revised Report and a definition of a language representation in terms of the ISO character set.

Wirth designed Pascal to satisfy two principal aims:

1. To make available a language suitable for teaching programming as a systematic discipline.
2. To define a language whose implementations would be both reliable and efficient on currently available computers.

Largely through the availability of a portable Pascal compiler, Pascal has been implemented on a large number of machines, particularly at teaching institutions.

By 1976, moves had been made to draw up an official international standard for the language. During that year, a Pascal Users Group (PUG) was formed to encourage the use of Pascal. In September 1977, the British Standards Institute (BSI) Pascal project got under way. By late 1978, the third working draft paper of the BSI committee was sent for processing. The paper was published in January 1979 [Pascal News 14 1979], by the Pascal Users Group for comment by Pascal users. Both ANSI and ISO announced, in late 1978, that committees had been formed to view the possibilities of

standardising Pascal.

It seems likely that a standard for Pascal will become available in the near future.

2.3 Structured Programming History and Background Information

2.3.1. History

Professor Edsger W. Dijkstra, of the University of Eindhoven, Netherlands, has been one of the driving forces in the movement towards structured programming. In a letter to the editor of the Communications of the ACM in March 1968, [Dijkstra 1968a], he suggested that the GOTO statement could be eliminated from programming languages, and claimed that the quality of a programmer was inversely proportional to the number of GOTO statements in his program. He also enunciated some of his ideas about top-down systems design (which seem to go hand in hand with the idea of structured programming) in a paper published later in that year [Dijkstra 1968b]. Dijkstra's objectives were to increase a programmer's programming ability by an order of magnitude and to discover the techniques (mental, organizational or mechanical) which could be applied in the process of program composition to achieve the increase.

Dijkstra suggested [Dijkstra 1972b] that a programmer should confine himself to intellectually manageable programs and, once he has done this, then develop a proof of his program before he writes the program itself. This approach goes some way in eliminating the debugging stage of program development. If a convincing proof is

first developed, and then a program written satisfying the structure of the proof, the correctness concerns turn out to be a very effective heuristic guidance.

IBM was experimenting with project organization at this time and, after a few initial tests, the company decided to try an experiment on a large scale: an information retrieval system for the New York Times [Baker 1972]. The major effort in this project was to develop an efficient new form of project organization and project management. However, structured programming and top-down programming also played a prominent part. The project was very successful and the productivity of the programmers involved was approximately five times higher than that of an average programmer. Since the successful completion of the New York Times project, the ideas used by IBM have spread throughout the industry.

A large number of universities and research organizations began experimenting with "goto-less" programming. In particular, workers at the Carnegie-Mellon University developed a "systems implementation language" called BLISS which does not have a goto statement. According to Professor W.A.Wulf [Wulf 1971], the experience of three years of BLISS and its use in the development of compilers and operating systems have shown it to be a practical and useful programming language.

In particular, Wulf reported [Wulf 1972] that

"The inescapable conclusion from the Bliss experience is that the purported inconvenience of programming without a goto is a myth. Programmers familiar with languages in which the goto is

present go through a rather brief and painless adaption period. Once past this adaption period they find that the lack of a goto is not a handicap; on the contrary, the invariant reaction is that the enforced discipline of programming without a goto, structures and simplifies the task".

A great deal of controversy has raged over the last few years concerning structured programming and "goto-less" programming. Mr Martin Hopkins summed up the situation by stating [Hopkins 1972]

"... our wisdom has not yet reached the point where future languages should eliminate the goto. If future work indicates that by avoiding goto we can gain some important advantage such as routine proofs that the programs are correct, then the decision to retain the goto construct should be reconsidered. But until then, it is wise to retain it".

Nevertheless, it seems that most of the data processing community is slowly becoming converted to structured programming. A number of books have appeared on the subject including [Dijkstra 1972a], [Greibach 1975], [Wirth 1973] and [Yourdon 1975] and a number of teaching institutions are now teaching a structured approach to programming. Computer scientists and research organizations are now striving to gain practical experience with structured programming and, at the same time, are looking for ways of using this programming approach to simplify the job of testing a computer program. It is expected that future research and experimentation will continue to demonstrate the greater productivity and control of programs written in this fashion.

2.3.2 The Translator's Interpretation of Structured Programming

The notion of structured programming is a philosophy of writing programs according to a set of rigid rules in order to decrease testing problems, increase programmer productivity and increase the readability of the resulting program. There are a number of programming aspects associated with the idea of structured programming and these include top-down program design, restricting the use of the GOTO statement by replacing it with other well-structured branching and control statements, the organization of a program's data into a logical structure and a number of other less important programming restrictions and conventions.

The translator attempts to eliminate as many FORTRAN GOTO statements as possible by introducing the structured control statements of Pascal (while .. do, repeat .. until, case .. of, if .. then .. else etc). By doing this, a number of unstructured sequences of statements are replaced by structured statements in Pascal and the readability of the resulting program is enhanced. The layout methods employed by the translator (see 5.3) add considerably to the clarity of the resulting program.

The translator attempts to impose a top-down style on a FORTRAN subprogram but its efforts in this direction are limited by the nature and style of the FORTRAN subprogram. The translator replaces unstructured FORTRAN sequences of statements by the structured statements of Pascal, but an unclear sequence of FORTRAN statements may not produce a clear structured Pascal sequence of statements.

The structured programming philosophy also involves the structuring of a program's data so that it may be dealt with in a more orderly manner. No attempt is made by the translator to impose a different interpretation on the FORTRAN program's data.

2.3.3 Theory and Techniques of Structured Programming

Top-down programming involves the design of a program as a nested set of modules with each module having a single entry point and a single exit. A paper by Bohm and Jacopini [Bohm & Jacopini 1966] has shown that such a structure can be composed from a language with only two basic control structures, the actual implementation of which is a function of the programming language. The concept presented in the Bohm and Jacopini paper, sometimes referred to as the "structure theorem", is of fundamental importance and is the basis for much of the implementation of structured programming. Their proof that any proper program can be constructed with the two basic control structures is therefore of great significance.

According to Bohm and Jacopini, we need three basic building blocks to construct a program:

1. A process block
2. A generalized loop mechanism
3. A binary-decision mechanism.

The process box, shown in Figure 2.1, may be thought of as a single computational statement, a machine language instruction, or any other computational sequence with only one entry and one exit - such as a subroutine.

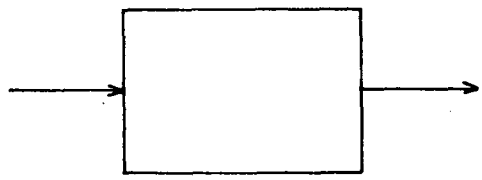


Figure 2.1 The Process Box

The loop mechanism, shown in Figure 2.2, is often referred to as a DO-WHILE mechanism and the binary-decision mechanism, shown in Figure 2.3, is often referred to as an IF-THEN-ELSE mechanism.

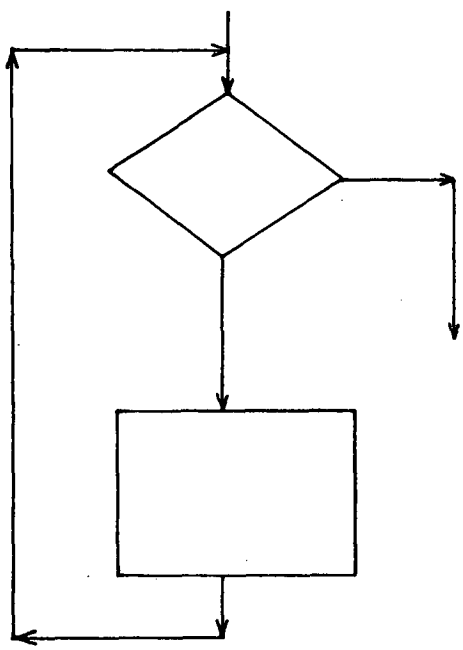


Figure 2.2 The Loop Mechanism

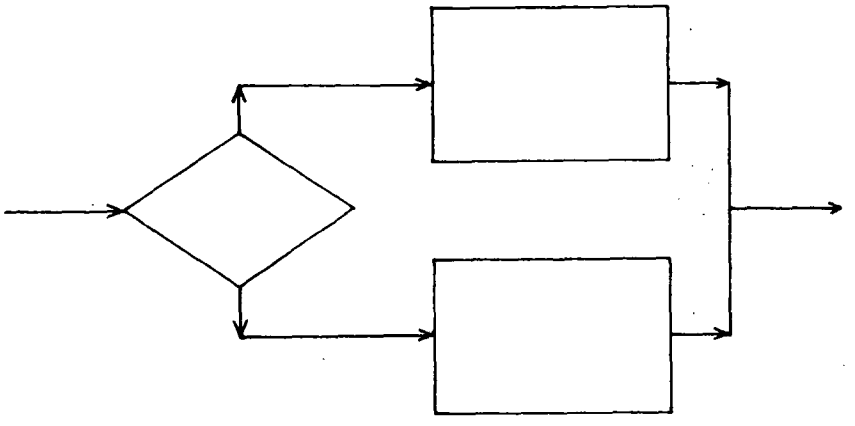


Figure 2.3 The IF-THEN-ELSE Mechanism

These three constructs can themselves be thought of as a process box since they each have only one entry and one exit. Thus, we can transform these constructs into process boxes and, by repeatedly applying this transformation, transform any program into a single process box. As Wulf [Wulf 1972] points out, this sequence of transformations may be used as a guide to understanding a program and proving its correctness. Conversely, the reverse sequence of transformations can be used to design the program in a top-down fashion, i.e., starting with a single process box, and gradually expanding it to a complex structure of the basic components.

2.4 FORTRAN Conversions

A number of people, including [Melton 1973] and [Prudom & Hennell 1977] have attempted the task of automatically converting FORTRAN programs into another language. Other people including [Meissner 1974] have proposed that the FORTRAN standard be altered to enable the language to accept some of the statements associated with structured programming. The recent publication of a draft FORTRAN standard [Fortran 1976] shows that a number of constructs of

structured programming have been proposed for the new standard.

Melton's work involved translating FORTRAN programs into structured extensions of FORTRAN, particularly IFTRAN, which has the two control constructs IF .. ELSE .. END IF and WHILE .. END WHILE. His translator automatically duplicated common code, or, when the code was too large to duplicate, converted the common code into a subroutine. Extra variables were only introduced to control multiple exits from loops. The techniques of the translator are sound and based on the ideas of Bohm and Jacopini [Bohm & Jacopini 1966]. The translator has the facility to produce richer languages than IFTRAN.

Prudom and Hennell began a project to automatically convert FORTRAN to Algol 68. The first part of this project involved a simple transliteration from FORTRAN to Algol 68. This method preserves the original logic of the FORTRAN program and minimizes the possibility of the introduction of errors. Restrictions were placed, initially, on the conversion of some statements and the principal restrictions are:

1. No input or output statements are converted.
2. All COMMON statements must be identical.
3. Only simple EQUIVALENCE statements are converted.

The authors report encouraging success with the transliteration and the resultant Algol 68 programs and the next version of their translator is designed to improve the program logic by the use of transformations.

3. FORTRAN TO PASCAL CONVERSION

3.1 Reasons for Translating FORTRAN to Pascal

In recent years, there has been an increasing interest in the Pascal language, particularly as more compilers and texts become available. One of the major deficiencies of the language definition is that it fails to recognise the vast quantity of important and useful software available in other languages. This blindness is also manifest amongst most compiler writers who fail to make provision for the linking of modules written in some other language. It is true, though, that a generalized interface would preserve some unattractive features, for instance, the user interface for FORTRAN routines with array parameters.

Users wishing to write in Pascal and also use an existing software library are faced with a dilemma. They must either change languages or recode these already existing algorithms into Pascal. It could be argued that these algorithms should be recoded in Pascal to benefit from the advanced data and logical structures available in the new language. Whilst this argument contains a large measure of truth, there are a vast number of algorithms which would not benefit from recoding (such as those which use only very simple data or logic structures). [Prudom & Hennell 1977] suggest that about half the contents of the NAG Numerical Algorithms Library falls into this category and the same result would probably apply to other numerical libraries.

Some Pascal compilers permit the inclusion of modules from another language as a non-standard extension. As standard Pascal

does not permit it, and the proposed new Pascal standard will not permit it, the only alternative to laboriously recoding these algorithms is to implement an automatic translation. Automatic translation could be implemented as transliteration which implies that each FORTRAN statement converts into an equivalent Pascal statement. This process is attractive since it preserves the logic structure of the original program and it suggests that the introduction of bugs into an otherwise thoroughly tested program is minimised. Documentation problems are also minimised since the original FORTRAN documents, with only fairly minor modifications would suffice. The advantages of translating a "GOTO" program to a "WHILE" program result from the fact that it is easier to understand what the structured version is meant to do. Translating "GOTO" programs is useful in maintaining, modifying, testing and documenting existing software. The structured version can replace the original or be used to refer back to statements in the original program. The advantage of performing the translation automatically is the efficiency of the operation and the reliability of the resulting "WHILE" program. The translated program performs the same algorithm as the original, and requires no additional debugging if the original program were correct.

FORTRAN and Cobol have become universally accepted languages and Rosen [Rosen 1972] claims that they will remain so in the future. He sees little reason to change to another language. Wirth [Wirth 1974] claims

> "The same inertia that kept many assembly code programmers from advancing to use FORTRAN is now the principal obstacle against

moving from a "FORTRAN" style to a structured style".
The existence of an automatic translator is an aid to overcoming the initial prejudice of a new language.

Yourdon [Yourdon 1975] regards FORTRAN as the one major high-level programming language (other than BASIC, perhaps) that is not suited to the concept of structured programming. He sees three major reasons for this:

1. There is no concept of block structures in FORTRAN, as it is presently constituted. That is, it is not possible to group several statements together and treat them as if they were one statement. This immediately forces the programmer to "jump" around groups of statements. While these GOTO statements are not evil in themselves, they begin to give the programmer the ability and the temptation to build complex structures.
2. There is no nested IF .. THEN .. ELSE in the FORTRAN language.
3. While there is a DO statement in FORTRAN, it is not as powerful as the analogous forms in Pascal. In FORTRAN, it is difficult to use the DO statement for anything other than iterative purposes, which is a common requirement in scientific applications. The concept of a DO .. WHILE is lacking.

While these points are a strong condemnation of the FORTRAN language, they do not necessarily mean that one cannot write structured programs in FORTRAN. The objective of structured programming is to decompose programs into smaller discrete units. There is no reason why this cannot be done with GOTO statements in FORTRAN but there is no guarantee that programmers would not use them

for any other purpose. Alternatively, it would be possible to develop an extended version of FORTRAN which would accept some block structures and use a preprocessor to convert those constructs to standard FORTRAN. Such action has been proposed by [Meissner 1974] and used by [Melton 1973].

For the time being, though, FORTRAN is a rather poor vehicle for structured programming concepts. The discipline that has to be imposed upon programmers to ensure a structured use of GOTO statements is difficult, if not impossible, to administrate.

3.2 Limitations on the Conversion of FORTRAN to Pascal

Despite the advantages which Pascal offers when it is compared to FORTRAN, there are some serious limitations in the Pascal language which hinder an automatic upward movement from FORTRAN to Pascal. This section looks at some of those limitations.

3.2.1 Input/Output

In FORTRAN I/O is record oriented. That is, each FORTRAN I/O statement accesses a new record on the external medium. In standard Pascal, I/O is stream oriented - files are considered to be one continuous stream of information and each I/O statement accesses the next available position in that stream.

A format specification may be used in a FORTRAN I/O statement. This specification may be identified by a statement label or an array identifier. In either case, it indicates that the variables to be read or written appear on the external medium according to the rules of the format specification and that each I/O statement causes a new

record to be accessed. Accessing of variables in the I/O list commences at the beginning of the new record, proceeds through that record and, maybe, to subsequent records.

In simple cases, where the number of variables in the I/O list is constant, it would be possible to translate FORTRAN formatted I/O to Pascal stream oriented I/O. This procedure would involve interpreting the type of each variable, matching it with the corresponding format specification and adjusting the format specification for any preceding blanks. The Pascal procedures readln and writeln would have to be used to force the Pascal program to skip to the end of the record after processing the last variable in the list.

There are a number of problems with this approach and the major ones are:

1. The number of variables in the I/O list must be fixed.
2. A (large) number of FORTRAN format specifications do not convert directly to Pascal stream oriented notation. Some of the problem cases are:
   (a) The FORTRAN E specification has no counterpart in Pascal.
   (b) Pascal prints the words TRUE or FALSE for boolean variables but in FORTRAN T or F are used.
   (c) The FORTRAN G specification depends on the run-time value of the corresponding variable for its format specification. Standard Pascal specifications must be natural numbers and specified at compile-time.

In practice, though, many FORTRAN I/O statements involve a

variable number of variables in the I/O list and the format specification may be repeated a number of times. Pascal insists that the number of variables in an I/O list is constant. It is not possible, without knowing the number of variables involved, to use the above approach for conversion.

Rather than issue an error message for each formatted I/O statement, the translator uses a non-standard feature of B6700 Pascal [B6700 Pascal 1978] to handle FORTRAN formatted I/O. B6700 Pascal permits the use of formatted I/O (see 4.5.2.8) and where a statement label is used in FORTRAN to indicate a FORMAT statement, the translator converts that statement to a Pascal formatted I/O statement.

If an array identifier is used to identify a format specification in FORTRAN, it implies that the format specification may vary during the execution of the program. Array identifiers are not permitted for format specifications in B6700 Pascal and the translator bars the translation of such specifications but prints a warning message at the offending statement in the FORTRAN listing.

If there is no format specification present in the FORTRAN I/O statement, the I/O action is performed in an unformatted mode. If a list is present in the I/O statement then the number of values used by the list may not exceed the number in the unformatted record. There is no direct equivalent statement in Pascal. The standard Pascal procedures read and write, as defined in the User Manual and Report, apply to textfiles only. However, the proposed new standard, printed in [Pascal News 14 1979], indicates that these procedures may

be used on non textfiles. The action of forcing each new I/O statement to start a new record is not available for non textfiles in Pascal, and, together with the problem of a variable sized I/O list, prohibit the conversion of unformatted I/O from FORTRAN to Pascal.

In FORTRAN, a file may consist of a combination of different types. The combination may be the same within each record or the type of the records may vary from record to record.

A Pascal file must be of a fixed type. This type may be one of the simple types - real, integer, etc, or it may be a record consisting of a combination of types. If a simple type is used, the data must be accessed sequentially but if a record type is used, the whole record must be accessed - individual components may not be accessed separately. Since the type must be fixed, it would be possible to implement variable types by using Pascal variant records but this could be a long, and messy process.

As the type of components of an unformatted file in FORTRAN is not fixed, in general, and the number of components may vary from record to record and the number of components written in a record may differ from the number read from that record, it is not possible to translate FORTRAN unformatted I/O statements into Pascal. The translator issues an error message when a FORTRAN unformatted I/O statement is encountered.

In general, the translation of I/O statements from FORTRAN to Pascal is unsolvable. The translator converts FORTRAN formatted I/O statements to use a non-standard feature of Pascal available in B6700

Pascal, but it makes no attempt to convert unformatted I/O.

3.2.2 Complex and Double Precision Types

In FORTRAN, a variable may be defined to be of COMPLEX or DOUBLE PRECISION type. Neither of these types is available in Pascal. A COMPLEX variable in FORTRAN is defined to be an ordered pair of real datum. It is possible to translate the complex type into a Pascal record of the form:

```
type
   COMPLEX = record
                RE : real;
                IM : real
             end;
```

Then each assignment to a complex variable would have to be translated to two Pascal assignments - one for the real part of the complex variable and the other for the imaginary part.

In FORTRAN a DOUBLE PRECISION variable is defined to occupy twice the number of storage units as a REAL variable. It is possible to translate a DOUBLE PRECISION variable to a real variable in Pascal, with a subsequent loss of accuracy. However, this transformation does not preserve the relationship of FORTRAN REAL variables to DOUBLE PRECISION variables. To preserve that relationship, it is necessary to declare a DOUBLE PRECISION type in Pascal:

```
type
   DOUBLEPR = record
                 D : real;
                 DUMMY : real
              end;
```

The FORTRAN variable would be converted to use the 'D' field of the record and the other field ('DUMMY') would be unused.

Both of these types in FORTRAN may be used to specify the type of a FUNCTION. However, in Pascal a function type must be a simple type or a pointer type. A structured type is illegal. Hence, a FORTRAN COMPLEX or DOUBLE PRECISION type function cannot be converted to a Pascal function.

3.2.3 Variable Arrays

In FORTRAN, it is possible to specify arrays whose sizes can only be determined at execution time. It is a common practice to use this facility in subprograms and the size of the variable arrays may vary each time the subprogram is called. In Pascal, each array must be declared with constant bounds - there is no facility for specifying bounds at execution time as there is in FORTRAN. There is no way, in general, of translating a FORTRAN variable size array into Pascal.

A number of proposals have been made to implement variable arrays in Pascal and various syntax styles are being investigated. It seems likely that variable arrays will become a part of the Pascal language in the near future.

4. CONVERSION DETAILS

4.0 General Comments

The following sections give the definitions which the translator follows for converting standard FORTRAN as defined by the Australian Standard 1486-1973 to Pascal as defined by the Pascal User Manual and Report [Jensen & Wirth 1975].

Where possible, one Pascal statement per FORTRAN statement is produced and the ordering of statements within each subprogram is preserved. However, the syntax differences between the two languages and the structural changes made by the translator necessitate that many FORTRAN statements move from their original order.

In general, it is not possible to preserve the arrangement of characters of a FORTRAN statement in Pascal. FORTRAN ignores spaces within identifiers but each Pascal identifier must be compact. Some FORTRAN functions are translated to Pascal functions whose names differ in length from the FORTRAN name. Some Pascal statements use a different syntax from their FORTRAN counterparts and the character arrangement within the statement varies accordingly. The layout of FORTRAN COMMENT statements is preserved in the translation to Pascal.

Some features, used in many versions of FORTRAN, but not standard, are accepted by the translator. These features are translated because of their usefulness and their widespread use.

Some features of Pascal are generated by the translator, even though they are not standard Pascal. These features are generated

because they are essential to the correct conversion of a FORTRAN program or because they are a useful, sometimes even a necessary, extension to standard Pascal.

4.1 Program Form

4.1.1 Character Set

FORTRAN defines a 47 character set consisting of A to Z, 0 to 9, 'blank',=,+,-,*,/,(,),',',. and $ (currency symbol). Pascal's character set includes all of these characters (and many more) but it does not define a currency symbol ($). FORTRAN defines the currency symbol but it is not mentioned again in the standard and so the translator makes no alteration to the character set.

4.1.2 Lines

A FORTRAN line consists of a string of 72 characters. The Pascal standard does not define the concept of a "line" of a program and so the translator performs the necessary translation on a line but preserves, where possible, the correspondence of one FORTRAN line to one Pascal "line". During the translation process, it may be necessary to reorder the program so that lines of Pascal do not appear in the same order as the FORTRAN source.

4.1.3 Comment Lines

The letter C in column 1 designates that line as a comment in FORTRAN. In Pascal, a comment is that piece of text enclosed between the symbols '{' and '}'. However, the Pascal standard allows for these symbols not being available on all systems by permitting '(*'

and '*)' to act as comment delimiters. The translator uses '(*' and '*)' as they are available on most systems and translates comments as follows :

(a) First Comment Line of a Group

If column 2 is a blank, minus, period or an asterisk then columns 1 and 2 are replaced by '(*'. If column 2 is not one of these characters then a new line is generated consisting solely of '(*' in columns 1 and 2. The FORTRAN comment line follows this line.

(b) Last Comment Line of a Group

If columns 71 and 72 both consist of blank, asterisk, minus or period characters then they are replaced by '*)'. If they do not consist of these characters then the comment line is printed followed by a line consisting of '*)' in columns 71 and 72.

In all cases the C in column 1 is replaced by a blank. The above process does not alter the layout of the text in any comment. Thus, any specific layout that the original writer may have wanted, is preserved.

If the first or last line of a comment group is a blank line (except for the C in column 1) then this line is omitted from the Pascal comment, but a blank line is printed in the Pascal listing. Blank lines in Pascal are allowable but are not used for any purpose apart from layout.

It is a common practice in FORTRAN to use blank comment lines to layout a subprogram or to break a subprogram into logical sections. If a group of FORTRAN comment statements is all blank then it is

translated into a series of blank lines in Pascal and the original layout of the FORTRAN program is preserved.

If the FORTRAN comment contained the character combination '*)' then errors could arise in a subsequent Pascal compilation as this sequence of characters delimits a Pascal comment and a compiler would treat further comments as valid non-comment data. If this combination does appear in a FORTRAN comment then the translator converts the '*' to a space character and prints a warning message to the user. This process preserves the parenthetical expression in the comment, allows a valid comment to be produced, and draws the user's attention to the problem encountered.

When a FORTRAN comment appears between two blocks of executable code, that comment is 'attached' to the block after the comment for reordering.

4.1.4 Continuation Lines

Pascal makes no allowance for continuation lines as FORTRAN does. Pascal assumes its input to be one continuous stream. The translator treats each statement as one single statement rather than an initial line plus continuation lines. Thus, any original layout in the FORTRAN statement is not preserved.

4.1.5 Statement Label

Statement labels in FORTRAN consist of 1-5 digits but in Pascal only a maximum of 4 digits is allowed. In the translation process most labels are eliminated because of the restructuring process.

Where it is necessary to use a label in the Pascal program the correspondence between labels is preserved for those labels not greater than 9999. However, for all 5 digit labels, a new label consisting of, at most 4 digits, is generated. To produce a new label, the translator uses the following method:

(1) Truncate the five digit label to four digits. If this new number is unique then it is used as the new label.

(2) If it is not unique, then 1 is added to the four digit label until a unique label is found. If a five digit number is reached before a unique label is discovered then 1 is subtracted from the original four digit number until a unique label is found. If zero is reached before a unique label is found, then an error message is printed. This is an extraordinary circumstance as it means that the subprogram contained 9999 labels and not one was used in conjunction with a FORMAT statement (see 4.5.2.8).

This process produces a label closely resembling the original label and some measure of correspondence with the original label is preserved.

eg. 45617 becomes 4561 or, if that is not unique then

4562, 4563, 4564, ... etc.

If the original label was a large number then the addition and then subtraction of 1 is used to preserve as much correspondence with the original label as possible.

eg. 99988 becomes 9998 but if that is not unique then

9999

9997, 9996, ... etc.

This method of producing a number is desirable, and better than

producing a random number, because it produces a number bearing some resemblance to the original 5 digit number.

The label on any FORMAT statement is not altered (see 4.5.2.8) and any label used on a FORMAT statement is available for reallocation as a statement label.

4.1.6 Identifiers

FORTRAN limits the size of its symbolic names to six alphanumeric characters, the first of which must be a letter. In Pascal, identifiers may be of any length but only the first 8 characters are significant. No FORTRAN identifier is altered by the translator. However, new identifiers are generated for boolean variables, format identifiers, etc. and the translator makes these identifiers greater than 6 characters in length to circumvent any problems of duplicate identifiers.

Most of the current Pascal compilers accept identifiers greater than 8 characters in length and the translator accepts a translator option, IDLENGTH (see 4.8.7), to allow the user to specify the maximum length of any identifiers to be generated. This option assumes a default value of 8, the standard length, and all identifiers generated by the translator are longer than 6 characters, contain not more than IDLENGTH characters and are unique over all IDLENGTH characters.

All identifiers generated by the translator have two parts:

(1) a series of letters, followed by

(2) a number (which may be empty).

A problem arises if the total number of characters in both parts exceeds the IDLENGTH parameter.

eg. FORMAT statements are translated into Pascal and an identifier of the form

FORMAT<nn>

is produced where <nn> is the statement label of the FORMAT statement (see 4.5.2.8). If nn were greater than 99 and IDLENGTH=8 then the identifier produced would be illegal as it would be larger than IDLENGTH characters. In cases such as this, the translator reduces the size of the string of letters, truncating from the right, until the number and letters form an identifier of IDLENGTH characters. An error is produced if an attempt is made to eliminate the first letter of an identifier.

eg. 99 FORMAT... produces an identifier FORMAT99

999 FORMAT... FORMA999

9999 FORMAT... FORM9999

These identifiers and others generated by the translator are unique because they are greater than 6 characters and, either the translator produces a unique number for each new identifier type produced (1,2,3,... etc.) or a label from the FORTRAN subprogram is used and FORTRAN requires each label to be unique within the subprogram in which it appears.

4.2 Data Types

4.2.1 Integer, Real and Boolean Types

Both FORTRAN and Pascal define these three data types and the definitions are almost identical. No problems are encountered in doing a direct conversion from FORTRAN to Pascal for these types.

4.2.2 Double Precision Type

FORTRAN defines a data type of double precision but Pascal does not define this data type (at least at this stage). The translator handles double precision types in one of two ways depending on the setting of the translator option switch, DOUBLE (see 4.8.4). If DOUBLE is set false, then the implication is that Pascal does not handle double precision types and the following statements are generated to appear in the Pascal outer block under the type heading:

```
DOUBLEPRECISION = record
                    D : real;
                    DUMMY : real
                  end;
```

All double precision variables are then declared to be records with two parts, both of which are real numbers. The first part is used to hold a Pascal real variable equivalent, as far as possible, to the FORTRAN double precision variable. The second part is not accessed but is used to preserve the FORTRAN relationship that a double precision variable occupies two storage units whereas a real variable occupies one.

By using this method, all FORTRAN double precision variables become Pascal real variables. Some loss of accuracy may occur and a warning message is issued when the above declaration is made.

The names D and DUMMY for the two parts of the double precision record were chosen for brevity. Each name need not satisfy the requirements of section 4.1.6 as the scope of each definition is the record itself.

The name of the record type will be DOUBLEPRECISION or as many letters from this name as the translator option, IDLENGTH, (see 4.8.7) permits. As IDLENGTH will not be less than 8, no problem of uniqueness will arise with the name.

If DOUBLE is set true, then an assumption that the version of Pascal to be produced handles double precision types is made and the syntax for handling double precision types is assumed to be similar to that of reals.

4.2.3 Complex Type

FORTRAN defines a complex type to be an ordered pair of real data - the first of the pair representing the real part and the second, the imaginary part. Pascal does not define this data type.

The translator handles the type COMPLEX in one of two ways depending on the setting of a translator option, COMPLEX (see 4.8.2). If COMPLEX is set false, then it is assumed that the version of Pascal to be produced does not handle the type complex and the following statement is generated to appear in the outer block of the

Pascal program under the type heading:

```
COMPLEX = record
            RE : real;
            IM : real
          end;
```

All COMPLEX declarations in the program then become Pascal records with two parts - one corresponding to the real part and the other to the imaginary part. For brevity, the identifiers RE and IM have been chosen to correspond to the two parts, real and imaginary respectively. These names need not correspond to the requirements of section 4.1.6 as their scope is limited to that of the record definition.

If the translator option is set true, then it is assumed that the Pascal compiler handles the complex data type. The syntax is assumed to be that of FORTRAN, ie. an ordered pair of real datum.

4.2.4 Hollerith Type

FORTRAN defines a Hollerith datum as a string of any characters capable of being represented on the processor. Pascal does not define a Hollerith type but defines a string as a sequence of characters. The translator converts all FORTRAN Hollerith data types to Pascal strings.

4.3 Data and Procedure Identification

4.3.1 Constants

4.3.1.1 Integer Constant

Both FORTRAN and Pascal define integer constants to be non-empty strings of digits and a direct conversion between the languages is possible.

4.3.1.2 Real Constant

FORTRAN defines a real constant to be written as an integer part, a decimal point, a decimal fraction and, optionally, a decimal exponent. Either the decimal fraction or the integer part may be empty but not both. If the decimal exponent is used the decimal point may be omitted if no decimal fraction is used.

The Pascal definition is similar except that each decimal string and the decimal point must appear in the real constant. However, in the case where an integer is raised to a power, the decimal point and the decimal fraction may be omitted.

The translator replaces an empty decimal string by a single zero.

eg. FORTRAN	Pascal Equivalent
10.32	10.32
99.	99.0
.357	0.357
1.47E10	1.47E10
2E10	2E10

4.3.1.3 Double Precision Constant

In FORTRAN, a double precision constant is written as a real constant, using the exponent form, except that the letter D is used instead of the letter E in the exponent part.

eg. 1.567D3

4D10

The translator translates double precision constants in one of two ways according to the setting of the translator option, DOUBLE (see 4.8.4).

If DOUBLE is set false, then all double precision variables and constants are converted to type real with, maybe, a subsequent loss of accuracy (see 4.2.2). The translator changes the double precision exponent indicator, D, to an E in Pascal.

If DOUBLE is set true, then no changes are made to the format of a double precision constant apart from replacing empty digit strings by zero (see 4.3.1.2). It is assumed that the version of Pascal to be produced handles double precision constants in the same format that FORTRAN uses.

4.3.1.4 Complex Constant

In FORTRAN a complex constant is written as an ordered pair of real numbers. The translator handles complex constants in one of two ways according to the setting of the translator option, COMPLEX (see 4.8.2).

If COMPLEX is set false, it is assumed that the version of Pascal to be produced does not recognise the type complex. A FORTRAN complex constant is translated to Pascal in accordance with the complex record definition of 4.2.3 and according to the statement in which it appears.

If COMPLEX is set true, an assumption is made that the version of Pascal to be produced recognises the data type, complex, and uses a format similar to that of FORTRAN for writing complex constants. In this case, only the rules for converting real constants (see 4.3.1.2) have to be satisfied during the translation.

4.3.1.5 Logical Constant

The logical constants in FORTRAN are written as .TRUE. and .FALSE.. The translator removes the periods surrounding these constants to convert them to their Pascal equivalents true and false. The Pascal syntax does not require the periods.

4.3.1.6 Hollerith Constant

In FORTRAN, a Hollerith constant consists of an integer constant, n, and the letter H followed by n characters. The translator changes the Hollerith constant into a Pascal string. This

necessitates surrounding the n characters by quotes (') and checking to see whether a quote character occurs in the string itself. If it does, it is duplicated because Pascal requires that if a quote occurs within a string, it appears twice.

4.3.2 Variables

In Pascal, all variables must be explicitly declared before being used. In FORTRAN, a variable may be explicitly declared at the beginning of a subprogram or it may be implicitly declared by being used in a FORTRAN statement.

The translator processes each declaration in FORTRAN and converts that statement to Pascal. The order in which DIMENSION and type declaration statements appear is preserved, as are any comments which are interspersed with these declarations. All other FORTRAN declaration statements cause some reordering in Pascal.

After all FORTRAN explicit declarations have been made in Pascal, the translator generates Pascal declarations for all those variables which are implicitly declared in the FORTRAN subprogram. The type declaration is in accordance with the type rule operating in the subprogram (see 4.3.5).

4.3.3 Arrays

In FORTRAN, arrays are restricted to a maximum of three dimensions and the subscript expressions are restricted in their format. No restrictions of this nature apply in Pascal and so each array and subscript expression may be converted directly to Pascal.

4.3.4 Procedures

Each procedure or function in Pascal must be specifically declared. The first pass of the translator picks out all procedure or function names and parameters and the second pass specifically declares all procedures and functions.

4.3.5 Type Rules

In FORTRAN, the type of a variable, array or function may be explicitly declared in a type statement. If an explicit declaration does not occur then the type depends on the first character of the name: I,J,K,L,M and N imply type integer; any other letter implies type real. In Pascal, each identifier must be explicitly declared. The translator generates explicit declarations for all implicit declarations of a FORTRAN subprogram.

The often used non-standard IMPLICIT statement (see 4.7.1) may be used to override the implicit declaration types in FORTRAN. The IMPLICIT statement is accepted by the translator and used in the declaration in Pascal of any implicitly declared FORTRAN variables.

4.4 Expressions

4.4.1 Arithmetic Expressions

In both FORTRAN and Pascal arithmetic expressions are formed with arithmetic operators and arithmetic elements. The arithmetic operators are :

FORTRAN Symbol	Denoting	Pascal Symbol
+	addition	+
-	subtraction	-
*	multiplication	*
/	division	/,div
**	exponentiation	(see below)

Table 4.1 Arithmetic Operators

The order of evaluation of arithmetic expressions in both languages is the same and follows the conventional mathematical line. There is no problem in converting the FORTRAN operators +, - and * to their Pascal equivalents. However, the operators / and **, representing division and exponentiation present a few problems.

In FORTRAN, the division operator, /, represents both integer and real division but in Pascal the symbol, /, represents real division and the operator div represents integer division. In FORTRAN, if the type of the two terms surrounding the division operator is integer then an integer division is performed, otherwise a real division is done. The translator converts all FORTRAN division operators to the Pascal real division operator, /, except in the above case where the div operator is used.

In Pascal, the exponentiation operator is not defined because of the problem of type involving the raising of an integer to a negative power

ie. I ** (-N)

FORTRAN treats this case as an error. However, the functions exp and ln are defined in Pascal and may be used to convert the FORTRAN exponentiation symbol to a Pascal expression. The arithmetic result

$$a^b = e^{b \ln a}$$

is used to convert the FORTRAN expression

A ** B

to the Pascal equivalent

exp (B * ln (A))

In Pascal, the type of this expression is always real but in FORTRAN the resultant type depends on the type of the elements involved.

A primary of any type may be exponentiated by an integer primary and the resultant factor is of the same type as that of the element being exponentiated. Thus, if both A and B are of type integer, the resultant factor is of type integer. As the above Pascal expression produces a real type, the Pascal expression has to be modified to produce an integer type. This is possible by using the round function:

round (exp (B * ln (A)))

However, this expression does not satisfy the case when B is negative eg. (I ** (-2)). Standard FORTRAN treats this case as an error and to include this case in the conversion to Pascal, the translator generates the following Pascal function, INTPOWER:

```
function INTPOWER (I1,I2 : integer) : integer;
var
   K : integer;
begin
   if ((I2<0) or ((I1=0) and (I2=0))) then HALT;
   K := 1;
   while (I2>0) do begin
      while not odd(I2) do begin
         I2 := I2 div 2;
         I1 := sqr(I1);
      end;
      I2 := I2 - 1;
      K := I1 * K;
   end;
   INTPOWER := K;
end;
```

This function satisfies all the FORTRAN requirements for raising an integer to an integer power. However, the function may be inefficient because it involves repeated multiplication and division until the power is reached, and, when it is used a warning message is produced by the translator highlighting the possible inefficiency.

A similar function, REALPOWER, is used to raise a real number to an integer power. If the translator option, DOUBLE, is set to true then the function DBLPOWER is used to raise a double precision number to an integer power.

When these functions are needed, they are declared in the outer

block of the Pascal program, before any other function or procedure declaration (see 4.6).

If the power involved in the FORTRAN factor is 2, then the Pascal function sqr will be used instead of one of the above functions.

eg. I ** 2 becomes sqr (I) in Pascal.

In FORTRAN, when the primary is of type real or double precision and the exponent is of type real or double precision, the resultant factor is real if both primaries are real, otherwise it is of double precision type.

If the translator option, DOUBLE, is true then the above convention is assumed to be true in Pascal. If DOUBLE is false, then all double precision variables are converted to type real and treated as described earlier in this section.

All other combinations of variable types in FORTRAN are illegal.

4.4.2 Relational Expressions

Both FORTRAN and Pascal have the same set of relational operators. However, the symbols representing the operators are different, as shown in Table 4.2.

FORTRAN Symbol	Representing	Pascal Symbol
.LT.	less than	<
.LE.	less than or equal to	<=
.EQ.	equal to	=
.NE.	not equal to	<>
.GT.	greater than	>
.GE.	greater than or equal to	>=

Table 4.2 Relational Operators

4.4.3 Logical Expressions

Logical expressions in FORTRAN and Pascal are formed from the three basic logical operators, and, or, and not. However, the symbols representing these operators are different in each language as shown in Table 4.3.

Logical Operator	FORTRAN Symbol	Pascal Symbol
negation	.NOT.	not
conjunction	.AND.	and
disjunction	.OR.	or

Table 4.3 Logical Operators

4.4.4 Evaluation of Expressions

In FORTRAN, when two elements are combined by an operator, the order of evaluation of the elements is optional provided that the mathematical laws of association and commutation are observed, and the integrity of parenthesized expressions is not violated. The

Pascal User Manual states that expressions in Pascal are evaluated from left to right according to the rules of parenthesis and operator hierarchy.

The rules for order of evaluation in FORTRAN are therefore freer than those of Pascal. An expression may be evaluated in one of many ways in FORTRAN. One of those ways is from left to right according to the rules of parenthesis and operator hierarchy ie. the rule for Pascal expression evaluation.

The FORTRAN standard states that a part of an expression need only be evaluated if such action is necessary to determine the value of the expression. Such action may be efficient in terms of execution time but could have side-effects through the non-evaluation of parts of an expression. The Pascal report makes no requirements on expression evaluation, but the User Manual states that it is optional whether all expressions are evaluated, or whether enough are evaluated to determine the result of the expression. The Pascal interpretation depends upon the compiler writers of each individual compiler. The choice made by the compiler writers will not affect the result of any expressions and the Pascal interpretation will yield the same results as the FORTRAN interpretation.

The definition of the terms simple expression and term vary only slightly between the two languages and a direct conversion between the two languages is possible. The definition of the terms expression and factor vary concerning the handling of relational expressions. In other aspects they are comparable and no conversion problems are encountered.

FORTRAN treats relational expressions as factors and processes an expression such as:

A .EQ. B .AND. C .EQ. D

as (A .EQ. B) .AND. (C .EQ. D)

Pascal, on the other hand, treats relational expressions as expressions, not factors, and would attempt to evaluate the above expression as:

A = (B and C) = D

creating an obvious error.

The translator overcomes this discrepancy between the two languages by placing parentheses around all relational expressions if no parentheses are present in the FORTRAN expression. Thus the above expression becomes in Pascal, after translation:

(A = B) and (C = D)

This method forces a Pascal compiler to treat the second relational expression as a factor and evaluate it before applying the logical operator (and). The value of the expression is not altered by this action.

Apart from the cases mentioned in this section, no other problems are encountered in converting FORTRAN expressions to Pascal.

4.5 Statements

In FORTRAN, statements are divided into two types - executable and non-executable. Executable statements specify actions but non-executable statements describe the characteristics and arrangements of data, editing information, statement functions and classification of program units. In Pascal, statements denote executable statements but the term declarations is used to correspond to non-executable statements in FORTRAN.

4.5.1 Executable Statements

4.5.1.1 Assignment Statement

In both FORTRAN and Pascal the assignment statement takes the general form

<variable><replacement operator><expression>.

The value of the <variable> is replaced by a new value specified by the <expression>.

The assignment rules for both languages vary according to the type of the variable and the expression, and those rules are summarised in Table 4.4.

<variable> type	<expression> type	FORTRAN rule	Pascal rule	FORTRAN to Pascal translator
integer	integer	assign	assign	direct conversion
integer	real	fix and assign	error	<variable> := trunc(<expression>)
integer	double precision	fix and assign	-	(see note (a)) (<variable> := trunc(<expression>))
integer	complex	error	-	-
real	integer	float and assign	assign	direct conversion
real	real	assign	assign	direct conversion
real	double precision	double precision evaluate & real assign	- (assign)	- (direct conversion)
real	complex	error	-	-
double precision	integer	double precision float & assign	- (assign)	- (direct conversion)
double precision	real	double precision evaluate & assign	- (assign)	- (direct conversion)
double precision	double precision	assign	- (assign)	- (direct conversion)
double precision	complex	error	-	-

<variable> type	<expression> type	FORTRAN rule	Pascal rule	FORTRAN to Pascal translator
complex	integer	error	-	-
complex	real	error	-	-
complex	double precision	error	-	-
complex	complex	assign	- (assign)	- (see note (b))
logical	logical	assign	assign	direct conversion

Table 4.4 Assignment Rules

Notes

(a) The assignment statement involving double precision expressions or variables depends on the setting of the translator option, DOUBLE (see 4.8.4).

If DOUBLE is false, then all double precision variables and expressions are converted to a type record (see 4.2.2) and the expression is evaluated according to the rule involving real instead of double precision. Each double precision variable is subscripted by ".D" to fit the record description.

If DOUBLE is true, then it is assumed that the version of Pascal to be produced handles the double precision type. The rules for handling this type are those indicated in the table in parentheses. These rules have been formed by extending the Pascal rules for type real. It has been assumed that the function trunc truncates double precision expressions to produce an integer result and that the

Pascal assignment operator converts an integer or real expression to a double precision type.

(b) The assignment statement involving complex expressions or variables depends on the setting of the translator option, COMPLEX (see 4.8.2).

If COMPLEX is set true, then it is assumed that the version of Pascal to be produced handles complex data types in a manner similar to that of FORTRAN. No special conversion procedures need take place for this data type.

If COMPLEX is set false, all FORTRAN assignment statements with either a complex variable or expression part are converted to two Pascal assignment statements - one for the real part and the other for the imaginary part. The assignment variable is converted to a record description with a field identifier used after the variable name to identify the part of the FORTRAN variable in each statement.

If a user defined complex function is encountered in the expression then a warning message is printed by the translator as complex functions are not converted to Pascal (see 4.6.3).

All combinations involving type LOGICAL not mentioned in the table are illegal in both FORTRAN and Pascal.

4.5.1.2 GO TO Assignment Statement

In FORTRAN the GO TO assignment statement is of the form

ASSIGN k TO i

where k is a statement label and i an integer variable. After

execution of such a statement, subsequent execution of any assigned GO TO statement using that integer variable causes the statement identified by the assigned statement label to be executed next, provided there has been no intervening redefinition of the variable.

There is no direct equivalent of this statement in Pascal. However, the translator replaces this statement by an assignment statement of the form

```
I := n
```

where n is a unique integer associated with each label in an ASSIGN statement to the variable, I. When an assigned GO TO statement is later encountered a case statement is used in Pascal (see 4.5.1.4).

Internally, in each subprogram, for each variable used, a list is constructed and each list element contains the statement label in FORTRAN and the corresponding integer to assign in Pascal. For example, consider the subprogram

```
ASSIGN 10 TO I
  .
ASSIGN 20 TO I
  .
ASSIGN 100 TO I
  .
```

The translator produces the Pascal equivalent

```
I := 1;
 .
I := 2;
 .
I := 3;
 .
```

and, internally, keeps a list which associates 1 with label 10, 2 with label 20, 3 with label 100, ...

In FORTRAN, it is possible to pass a variable, used in an ASSIGN statement, to a subprogram and to use that variable in an assigned GOTO statement in the subprogram, or subsidiary subprograms - maybe as an error exit. No equivalent action is possible in Pascal and where such an action occurs, the translator issues a warning message.

4.5.1.3 Unconditional GO TO Statement

Both FORTRAN and Pascal use a go to statement of the form

GO TO <label>

In FORTRAN, the <label> must appear in the same program unit as the GO TO statement but in Pascal it may be in an outer block or procedure. The Pascal definition is, therefore, wider than that of FORTRAN but no problems are encountered in performing a direct conversion from FORTRAN.

4.5.1.4 Assigned GO TO Statement

In FORTRAN, an assigned GO TO statement takes the form

$$\text{GO TO } i,(k_1,k_2, \ldots, k_n)$$

where i is an integer variable and each k a statement label. At the time of execution of an assigned GO TO statement, the current value of i must have been assigned by the previous execution of an ASSIGN statement to be one of the statements in the parenthesized list, and such an execution causes the statement identified by that statement label to be executed next.

There is no equivalent statement in Pascal. However, by using the list generated for the ASSIGN statements (see 4.5.1.2) a case statement may be used. Each label used in the ASSIGN statements is associated with a unique integer and this integer may then be used to select a path in a case statement.

eg. The FORTRAN program

```
      ASSIGN 10 TO I
        .
      ASSIGN 20 TO I
        .
      GO TO I, (10,20)
   10   .
        .
   20   .
        .
```

becomes, in Pascal

```
I := 1;
   .
I := 2;
   .
case I of
1: .     {group of statements starting with label 10}
   .
2: .     {group of statements starting with label 20}
   .
end;
```

If, in FORTRAN, an ASSIGN statement appears more than once assigning the same label to the variable, then, in Pascal, there are two ways to handle the situation.

If the FORTRAN program was

```
ASSIGN 20 TO I
   .
ASSIGN 20 TO I
   .
GO TO I, (10,20)
   .
```

then the Pascal program could become

```
    .
I := 2;
    .
I := 3;
    .
case I of
1: .     {group of statements starting with label 10}
    .
2,3: .   {group of statements starting with label 20}
    .
end;
```

or

```
    .
I := 2;
    .
I := 2;
    .
case I of
1: .  {group of statements starting with label 10}
    .
2: .  {group of statements starting with label 20}
    .
end;
```

The translator produces a Pascal program using the style of the second method in this case because that method more closely resembles the original FORTRAN program and it saves the generation of a possible large number of case labels.

4.5.1.5 Computed GO TO Statement

In FORTRAN, a computed GO TO statement is of the form

GO TO (k_1,k_2, ... k_n),i

where each k is a statement label and i is an integer variable. The Pascal equivalent of this statement is the case statement:

```
case i of
1:  .   {statements whose initial label was k1}
    .
2:  .
    .
n:  .   {statements whose initial label was kn}
    .
end
```

In both languages, the effect is undefined if the value of i lies outside the range 1 to n, so no checks to that effect are inserted by the translator.

If a label appears more than once in a computed GO TO statement then more than one label will appear on the corresponding block of code in the Pascal program. A Pascal case label will be generated for each appearance of the FORTRAN label. This approach saves the repetition of code when a FORTRAN label is used more than once.

eg. The FORTRAN statement

```
GO TO (100,100,101,100,101),I
```

becomes in Pascal

```
case I of
1,2,4:  .   {statements whose initial label was 100}
        .
3,5:    .   {statements whose initial label was 101}
        .
end
```

4.5.1.6 Arithmetic IF Statement

In FORTRAN, the arithmetic IF statement is of the form

IF (<arithmetic expression>) k_1,k_2,k_3

where the expression is of integer, real or double precision type and each k is a statement label. This statement is a three-way branch, and execution of the statement causes the evaluation of the expression, following which the statement identified by the label k_1, k_2 or k_3 is executed next as the value of the expression is less than zero, equal to zero or greater than zero respectively.

There is no direct equivalent of this statement in Pascal. However, an equivalent sequence of statements may be constructed using the Pascal if statement as follows:

```
<variable> := <arithmetic expression>;
if (<variable> < 0) then begin
   .
   .    {block of statements starting with label k1}
   .
end else begin
   if (<variable> = 0) then begin
      .
      .    {block of statements starting with label k2}
      .
   end else begin
      .
      .    {block of statements starting with label k3}
      .
   end;
end
```

In the general case, as outlined above, a temporary variable is needed to hold the value of the arithmetic expression. This method of evaluating the expression first, before doing a test, saves the expression from being evaluated twice, and thus any side effects from the second evaluation.

A number of modifications may be easily made to the general outline :

(1) If the arithmetic expression is a simple variable or array element then there is no point in allocating a temporary variable for use by the arithmetic IF statement - the tests are performed on the simple variable or array element directly.

(2) If two of the statement labels are identical then the general outline reduces to a Pascal if...then...else statement and there is no need for a temporary variable to hold the value of the arithmetic expression.

If the FORTRAN expression was:	The Pascal equivalent is:
IF (<a.e.>) k_1,k_1,k_2	if (<a.e.> <= 0) then begin . {statements beginning with . label k_1} end else begin . {statements beginning with . label k_2} end
IF (<a.e.>) k_1,k_2,k_1	if (<a.e.> = 0) then begin . {statements beginning with . label k_2} end else begin . {statements beginning with . label k_1} end
IF (<a.e.>) k_1,k_2,k_2	if (<a.e.> < 0) then begin . {statements beginning with . label k_1} end else begin . {statements beginning with . label k_2} end

For each of these cases equivalent expressions exist but the

above expressions are desirable because they preserve the left to right nature of FORTRAN (cases 1 and 3) and use positive logic (case 2).

The layout of the statements following the arithmetic IF statement may not be preserved. The first executable statement following an arithmetic IF statement must be labelled, otherwise it would never be reached in an executable sequence. In practice, this statement is usually one of the branches of the arithmetic IF, but this is not always the case. The translator re-orders the blocks following a FORTRAN arithmetic IF statement so that, in Pascal, the blocks associated with each branch of the if statement form part of the if...then...else... sequence. The original order of statements in FORTRAN may not be preserved in this case.

When a temporary variable needs to be allocated, the translator uses the names

ARMIF01, ARMIF02, ... etc. (for ARithMetic IF)

to easily identify these variables with arithmetic IF statements and to prevent duplicate identifiers (see 4.1.6).

4.5.1.7 Logical IF Statement

In FORTRAN, a logical IF statement is of the form

IF (<e>) S

where e is a logical expression and S is any executable statement except a DO or another logical IF. Upon execution of this statement, the expression is evaluated and if it is true the statement S is executed. Execution continues at the next statement if the

expression is false.

The Pascal equivalent of this statement is of the form

if (<e>) then S

There are no problems with a direct FORTRAN to Pascal conversion for this statement. If statement S is a RETURN, STOP or GO TO statement then the format of the statement S may change in Pascal to that specified by the translator (see 4.5.1.3, 4.5.1.4, 4.5.1.5, 4.5.1.9, 4.5.1.11) for these statements.

4.5.1.8 CALL Statement

The CALL statement in FORTRAN is of the form

CALL S(a_1,a_2, ...,a_n)

or CALL S

where S is the name of a subroutine and each a is an actual argument. The Pascal equivalent of this statement is the procedure statement which takes the form

S (a_1,a_2, ... ,a_n)

or S

There are no problems in translating the FORTRAN CALL statement to the Pascal procedure statement. However, see 4.6.4 for a discussion on argument conversion.

In Pascal, a procedure may be entered more than once before the first reference has been exited but in FORTRAN no recursion is allowed - a procedure subprogram may not be referenced twice without a RETURN statement in that procedure having intervened. The Pascal definition for procedure entry is wider than that of FORTRAN but no

conversion problems are encountered in this area.

4.5.1.9 RETURN Statement

A RETURN statement in FORTRAN is of the form

RETURN

and it marks the logical end of a procedure subprogram. In Pascal, no equivalent statement exists as the only way to exit a procedure is to reach the end of the outermost block of that procedure (ie. the last end).

The translator translates a FORTRAN RETURN statement into a Pascal goto and generates a label, if necessary, to be placed in front of the last end of the Pascal procedure.

eg. FORTRAN	becomes	Pascal
SUBROUTINE S		procedure S;
		label n;
.		.
.		.
RETURN		goto n;
.		.
.		.
END		n: end;

If a RETURN statement is immediately followed by an END statement in FORTRAN, then the above steps will not be performed as, in Pascal, they will be done by definition.

4.5.1.10 CONTINUE Statement

In FORTRAN, a CONTINUE statement is of the form

CONTINUE

and upon execution causes continuation of the normal execution sequence. There is no equivalent statement in Pascal and the translator replaces it by the Pascal empty statement. However, during the reorganization of the FORTRAN program, most of the Pascal empty statements are dropped. The CONTINUE statement is largely used in FORTRAN to signify the end point of a DO loop. In this case, the translator does not use the empty statement in Pascal but absorbs the CONTINUE into the construct replacing the FORTRAN DO (see 4.5.1.13).

4.5.1.11 STOP Statement

The STOP statement in FORTRAN takes the form

STOP n

or STOP

where n is a string of 1-5 octal digits. Execution of this statement causes termination of execution of the program. There is no direct equivalent statement in Pascal because the cessation of execution of a Pascal program is only achieved when the end of the outermost block is reached. In FORTRAN, a STOP statement may occur wherever an executable statement may be.

If the STOP statement occurs as the last statement before the END of the main program, the translator ignores the statement. The stopping effect will be achieved by definition. However, if the STOP occurs anywhere else in the main program or a subprogram it is dealt with in a manner similar to the RETURN statement (see 4.5.1.9). ie. a label is allocated to be placed in front of the end. of the Pascal program and a STOP statement is translated into a goto that label.

ie. FORTRAN	becomes	in Pascal
		program ....;
		label n;
.		.
STOP		goto n;
.		.
.		.
END		n: end.

It makes no difference in Pascal whether the FORTRAN STOP was in the main program or a subprogram as all subprograms are bound into the scope of the main Pascal block. The octal digits, if present in the FORTRAN STOP statement, are ignored by the translator.

4.5.1.12 PAUSE Statement

The PAUSE statement, in FORTRAN, is of one of the forms

PAUSE n

or PAUSE

where n is a string of 1-5 octal digits. The inception of execution of this statement causes the cessation of execution of an executable program. Execution is resumable but the decision to resume is not under the control of the program. At the time of cessation the octal digit string is accessible and, if execution is resumed, the completion of the PAUSE statement causes continuation of the normal execution sequence.

There is no direct equivalent of this statement in Pascal. With the advent of sophisticated operating systems the PAUSE statement in

FORTRAN is one statement which has almost become obsolete. Its main use occurred in the days of primitive operating systems when, usually, the programmer was running his own program and he could use the PAUSE to initiate or indicate that some operator action was required or he could use it to halt his program so that he could examine the state of his program. In that case it was a handy debugging tool. In most modern day sophisticated operating systems a programmer does not control the execution of his program. The operating system handles it for him and he has no direct interaction with his program's execution.

These sophisticated operating systems handle the FORTRAN PAUSE statement in one of two ways :

(1) ignore it completely and resume the normal execution sequence (CDC 3300 MASTER operating system)

(2) make the octal digits available to the job control language so that it may decide what action to take and when to resume execution (ICL 1904A GEORGE II and III operating systems and Burroughs B6700 MCP operating system).

As there is no way of communicating with the job control language in Pascal, the translator takes the former action and ignores the translation of the PAUSE statement completely, except that a warning message is issued when a PAUSE statement is encountered.

4.5.1.13 DO Statement

The DO statement in FORTRAN takes the form

$$\text{DO n i} = m_1, m_2, m_3$$

Parameter m_3 is optional and when it is omitted (with its preceding comma) the value 1 is assumed.

n is the statement label of a statement that physically follows the DO statement in the same subprogram. The statement labelled n is called the terminal statement. i is an integer variable name, called the control variable. m_1 is the initial parameter, m_2 the terminal parameter and m_3 the incrementation parameter and each is an integer constant or an integer variable reference. Each m must be greater than zero.

A DO statement is used to define a loop and associated with each DO is a range that is defined to be those executable statements from and including the first executable statement following the DO to and including the terminal statement associated with the DO.

The action of a DO statement is summarized in Figure 4.1.

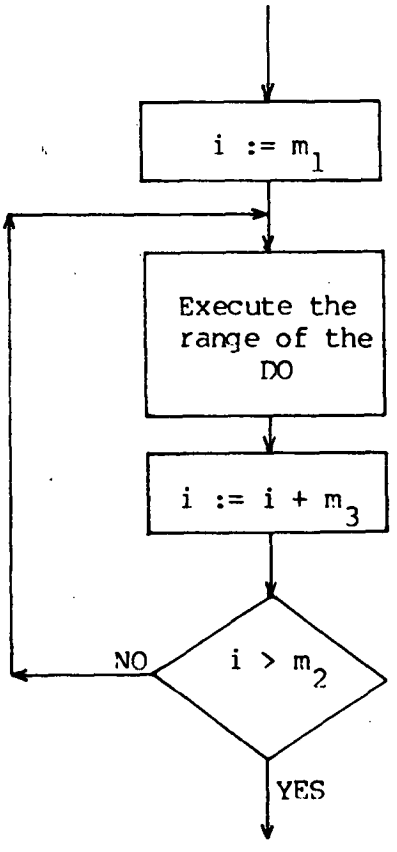


Figure 4.1 DO statement

There is no direct equivalent of this statement, in general, in Pascal. However, the DO statement is easily translated into a number of Pascal statements. In the general case, the following Pascal statements are equivalent to a FORTRAN DO statement and its range.

```
i := m1;
repeat
   .
   .      {Range of DO statement}
   .
  i := i + m3;
until (i>m2)
```

If the DO statement in FORTRAN has an incrementation parameter

of 1 and the initial value is guaranteed less than or equal to the final value then a for statement may be used in Pascal to give a neater translation.

```
for i := m1 to m2 do
begin
  .
  .     {Range of DO statement}
  .
end
```

In practice, many FORTRAN DO statements have an incrementation parameter of 1, and the second parameter larger than the first when the statement is executed so that the loop is executed at least once.

ie. statements such as

```
    DO 10 I=1,N        (N>1)
or  DO 20 J=M,N        (N>M)
```

As the bounds of the DO loop are variables, the translator must issue a general translation and use the repeat format rather than the for statement. A translator option, FORSTMT (see 4.8.6), exists and it allows the user to specify that he wants all DO statements with an incrementation parameter of 1 and at least one other parameter not a constant, to be converted into Pascal for statements. The user is then responsible for seeing that the loop is executed at least once, as no check to this effect is generated by the translator. Knuth [Knuth 1971] claims that 95% of FORTRAN DO statements use an incrementation parameter of 1, and have the second parameter larger than the first.

4.5.1.14 Input and Output

In FORTRAN, an input/output unit is identified by an integer value which may appear in a statement as an integer constant or an integer variable reference. In Pascal, file variables are represented by identifiers. The translator converts a FORTRAN I/O unit designator, represented by an integer constant, into an identifier of the form

FORFILEn

where n is the unit designator in FORTRAN (see 4.1.6).

An I/O unit in FORTRAN may also be represented by an integer variable whose contents, during the execution of the program, may vary to signify that future I/O actions are to be performed on a different file. In Pascal, each file is identified by an identifier and there is no facility for changing the association between a file and its identifier during the execution of a program. When a FORTRAN program uses this facility the translator prints a warning message in the FORTRAN listing and performs no further translation on the unit identifier.

When a file unit identifier is used in FORTRAN, the translator declares that identifier in the outermost block of the Pascal program, regardless of where the file is used in the FORTRAN program. Files in FORTRAN are considered to be global entities and may be accessed from any subprogram in the FORTRAN program. Each file is considered to be a sequence of logical records and each I/O statement accesses the next logical record in the sequence (except for the BACKSPACE statement).

No FORTRAN statement is capable of creating or removing a file or altering its characteristics. These functions are external to the FORTRAN program but fit in with the global concept of FORTRAN files.

In Pascal, files have the same scope of definition as all other variables, ie. they are local to the block in which they are defined. Thus, if two independent subprograms need access to a file, that file must be declared to be global to both subprograms. The mode of access is assumed to be sequential. In Pascal, however, subprograms may declare files, access them, and at the end of the subprogram that definition is removed along with all other definitions made in that block. If a file is declared in the outermost block, it is global to all subprograms and any subprogram may access the file.

To overcome any possible problem of Pascal subprograms accessing files, each file declaration is made in the outermost block of the Pascal program.

4.5.1.14.1 READ and WRITE Statements

The READ and WRITE statements in both languages specify the transfer of information. Each statement may include a list of names of variables, arrays and array elements.

Records may be formatted or unformatted. The transfer of a formatted record requires that a format specification be referenced to supply the necessary positioning and conversion specifications. The number of records transferred by the execution of a formatted READ or WRITE is dependent upon the list and referenced format

specification. A format statement in Pascal is non-standard but is permitted in the Burroughs B6700 version of Pascal. It is a useful extension to standard Pascal and necessary to give a translation of FORTRAN I/O statements in Pascal. All further discussion on formats in Pascal should be regarded as non-standard.

4.5.1.14.1.1 Input/Output Lists

An input/output list in FORTRAN specifies the names of the variables and array elements to which values are assigned on input or whose values are transferred on output.

A list is a simple list, a simple list enclosed in parentheses, a DO implied list, or two lists separated by a comma. A simple list is a variable name, an array element or an array name, or two simple lists separated by a comma.

There is no problem in translating a variable name or an array element name directly to Pascal. However, the syntax of Pascal does not allow the appearance of an array name in an I/O list. When an array name appears in FORTRAN, it specifies all of the array element names defined by the array declarator. Therefore, in Pascal, an array name has to be translated into a for statement as follows :

for DOIMP00:=1 to <max. array size> do A[DOIMP00]

where the original FORTRAN program merely specified A in an I/O list. Note that Pascal requires the allocation of a loop control variable for the for statement and the names

DOIMP00, DOIMP01, ... ["DO IMPlied"]

are used by the translator to uniquely identify these variables in accordance with the arguments of 4.1.6.

If the array in the original FORTRAN program was multidimensional then the conversion would produce a Pascal I/O list of the form

```
for DOIMP02:=1 to <max.dimension 2> do
   for DOIMP01:=1 to <max. dimension 1> do
      A[DOIMP01,DOIMP02]
```

A DO implied list is a list followed by a comma and a DO implied specification, all enclosed in parentheses. A DO implied specification is of the form

$i = m_1, m_2, m_3$

or $i = m_1, m_2$

where the elements i, m_1, m_2 and m_3 are defined as for a DO statement (see 4.5.1.13).

As for a DO statement, the translator converts a DO implied specification into one of the following forms :

```
begin
   i := m1;
   repeat
      .
      .   {Do implied list}
      .
      i := i + m3;
   until (i>m2)
end;
```

or

```
for i:=m1 to m2 do {DO implied list}
```

See 4.5.1.13 for a discussion on the two forms of translations.

The translations shown above for DO implied specifications are not standard Pascal but they are permitted in B6700 Pascal and these facilities are a useful extension to the Pascal report and necessary for the successful conversion of FORTRAN I/O to Pascal.

4.5.1.14.1.2 Formatted READ Statement

A formatted READ statement in FORTRAN is of one of the forms

READ(<unit>,<format specification>) <list>

or READ(<unit>,<format specification>)

The translator converts these FORTRAN statements into Pascal statements of the form

read(FORFIL<unit>,FORMAT<format label>,<list>)

or read(FORFIL<unit>,FORMAT<format label>)

4.5.1.14.1.3 Formatted WRITE Statement

A formatted WRITE statement in FORTRAN is of one of the forms

WRITE(<unit>,<format specification>) <list>

or WRITE(<unit>,<format specification>)

The translator converts these FORTRAN statements into Pascal statements of the form

write(FORFIL<unit>,FORMAT<format label>,<list>)

or write(FORFIL<unit>,FORMAT<format label>)

4.5.1.14.1.4 Unformatted READ Statement

An unformatted READ statement in FORTRAN is of one of the forms

READ (<unit>) <list>

or READ (<unit>)

Execution of this statement causes the input of the next record from the unit and, if there is a list, the values are assigned to the sequence of elements specified by the list. The sequence of values required by the list may not exceed the sequence of values from the unformatted record.

As stated in 3.2.1, it is not possible, in general, to convert this statement to Pascal and the translator issues an error message when an unformatted READ statement is encountered.

4.5.1.14.1.5 Unformatted WRITE Statement

In FORTRAN, an unformatted WRITE statement is of the form

WRITE (<unit>) <list>

Execution of this statement creates the next record on the <unit> from the sequence of values specified by the <list>.

For reasons similar to those concerning the unformatted READ statement (see 4.5.1.14.1.4) this statement is not translated to Pascal but an error message is issued.

4.5.1.14.2 Auxiliary Input/Output Statements

4.5.1.14.2.1 REWIND Statement

In FORTRAN, a REWIND statement is of the form

REWIND u

Execution of this statement causes the unit identified by u to be positioned at its initial point.

In standard Pascal, in general, there is no equivalent statement. The Pascal procedures reset and rewrite perform a rewind function but they also perform other actions which make their use, in general, unacceptable as an equivalent statement to FORTRAN's REWIND.

In B6700 Pascal, the statement

close (<file>,rewind)

is acceptable as an equivalent statement to the above FORTRAN statement. This statement closes the file as well as rewinding it. If the file is used again, later in the program, the next I/O action on that file will re-open it. The FORTRAN statement does not close the file - it merely repositions it. The closing and opening action in B6700 Pascal will add a small overhead to the converted program but it preserves the basic FORTRAN function.

If the B6700 translator option is set, then the B6700 close statement will be generated by the translator. Otherwise, a comment

(* REWIND STATEMENT WAS HERE *)

will be inserted into the Pascal program.

4.5.1.14.2.2 BACKSPACE Statement

A BACKSPACE statement in FORTRAN is of the form

BACKSPACE u

Execution of this statement results in the positioning of the unit, u, so that what had been the preceding record prior to that execution becomes the next record.

There is no equivalent statement in Pascal and, where this statement is used, a warning message in the FORTRAN listing is printed by the translator.

4.5.1.14.2.3 ENDFILE Statement

An ENDFILE statement in FORTRAN is of the form

ENDFILE u

Execution of this statement causes the recording of an endfile record on the unit identified by u.

There is no equivalent statement in standard Pascal. However, in the B6700 version of Pascal, a close statement may be used to perform the same function. The translator converts all ENDFILE statements into close statements of the form

close(FORFIL<u>)

if the B6700 translator option has been set. Otherwise, it prints a comment

(* ENDFILE STATEMENT WAS HERE *)

in the Pascal listing.

4.5.1.14.3 Printing of Formatted Records

When formatted records are prepared for printing in FORTRAN, the first character of the record is not printed but is used to determine the vertical spacing on the output file.

This method of spacing is not available in standard Pascal. Some versions of Pascal have facilities to permit vertical spacing but no general procedure is available.

The translator drops the first character from a FORTRAN FORMAT statement used for output, and the first non-slash character after a slash within the FORMAT statement. The output from the corresponding Pascal program may not be the same as that from the FORTRAN program because, in general, it is not possible to detect the first character of second and subsequent lines and drop that character.

4.5.2 Non-Executable Statements

4.5.2.1 Array Declarator

In FORTRAN, an array declarator specifies an array used in a program unit. It may be used in a type statement, DIMENSION or COMMON statement, and it indicates the name, number of dimensions and the size of each dimension. It takes the form

v (i)

where v is the symbolic name and (i) is composed of 1, 2 or 3 expressions, each of which may be an integer constant or integer variable name.

In Pascal, the concept of arrays is wider than that allowed in

FORTRAN. The general array declaration takes the form

array [T1] of T2

where T1 is a scalar type (not integer or real) and T2 is a component type. The bounds of the Pascal array are therefore fixed by this definition and cannot be altered during the execution of a program. The range of values allowed for subscripts is wider in Pascal (eg. negative values may be used) but the range employed by FORTRAN (1 to n) will easily convert to Pascal. Pascal sets no limit on the number of dimensions of an array.

In general, the FORTRAN declarator

v (i)

becomes, in Pascal

array [1..n, 1..m, 1..l] of <type of v>

where n is the high bound of the first dimension, and m and l the high bounds of the second and third dimensions. If there is no second or third dimension then their declaration and the preceding comma are omitted. In FORTRAN, if any of the entries in the declarator subscript is an integer variable name, the array is called an adjustable array. Such an array may only appear in a procedure subprogram and the dummy argument list must contain the array name and the integer variable names that represent the adjustable dimensions. For every array appearing in an executable program there must be at least one constant array declarator associated with it through subprogram references.

In Pascal, array bounds must be constant in both the main program and procedures. In general, therefore, it is not possible to convert FORTRAN variable array declarators to Pascal because each

call to a subprogram may specify a different size for the variable array. The translator issues an error message when a variable array declarator is encountered in the FORTRAN source program.

4.5.2.2 DIMENSION Statement

In FORTRAN, a DIMENSION statement is of the form

DIMENSION $v_1(i_1),v_2(i_2)$, ...

where each v(i) is an array declarator.

The equivalent statements in Pascal take the form

v_1 : array [<array bounds>] of <v_1 type>;

v_2 : array [<array bounds>] of <v_2 type>;

.

.

where each <array bounds> takes the form

1 .. <max bound for dimension 1>,1 .. <max bound for dimension 2>,..

to a maximum of three dimensions.

eg.

FORTRAN	Pascal
DIMENSION A(6),J(2,3)	A : array [1..6] of real;
	J : array [1..2,1..3] of integer;

Note that one FORTRAN statement may declare many arrays of different sizes and that each Pascal array declaration may declare many arrays but all of the same size.

4.5.2.3 COMMON Statement

In FORTRAN, a COMMON statement is of the form

COMMON $/x_1/a_1/$ ... $/x_n/a_n$

where each a is a non-empty list of variable names, array names or array declarators and each x is a symbolic name or is empty. Each x is a block name that bears no relationship to any variable or array of the same name.

The translator transfers all COMMON blocks to the outermost block of the Pascal program and creates a Pascal record structure from the FORTRAN COMMON block. The COMMON block name becomes the record name and each variable used in a COMMON block is prefixed by the record name in Pascal when used in a statement.

Thus the FORTRAN statements

```
COMMON /CBLK/ X,I,A(10)
  .
  .
X=A(1)
I=3
  .
```

become, in Pascal

```
CBLK : record
         X : real;
         I : integer;
         A : array [1..10] of real
       end;
  .
  .
CBLK.X := CBLK.A[1];
CBLK.I := 3;
  .
```

where the record, CBLK, has been transferred to the outer block. The name of the COMMON block must be unique in the subprogram in which it is used in Pascal. If it is not unique in the FORTRAN program, the translator adds as many letters from the name "COMMON" to the name as is permitted by 4.1.6. The name BLANKCOMMON is used for a Pascal record of a FORTRAN blank COMMON block, in accordance with 4.1.6.

In FORTRAN, it is possible to use different names for COMMON variables in each subprogram.

eg. In subprogram 1

```
COMMON /CBLK/ X,I,A(10)
```

in subprogram 2

```
COMMON /CBLK/ X,I,B(10)
```

These two COMMON statements refer to the same storage area but a different identifier is used in each subprogram to access the COMMON array. The translator uses a variant record, in Pascal, for this situation and translates the above example to

```
CBLK : record
         X : real;
         I : integer;
         case CBLKINTE : boolean of
         true: (A : array [1..10] of real);
         false: (B : array [1..10] of real)
       end;
```

where CBLKINTE is a boolean variable which is only used, in this case, as a dummy variable to satisfy the syntax of Pascal. The translator adds the word INTEGER to the common block name to make it unique and uses the name for the variable (see 4.1.6). If there are

more than two variant parts of the record then CBLKINTE is declared to be of type integer and the labels 1, 2, ... etc. are used instead of true and false.

In Pascal, however, a variant part is possible only at the end of the record description. It is not possible to have the variant part at the beginning of the record description and the fixed part at the end. For this reason, some identifiers in the Pascal record description have to be altered to make them unique within the record description. The translator attaches integers to an identifier to make it unique (see 4.1.6).

eg. If the COMMON statements

```
COMMON /CBLK/ X,I,A(10)
COMMON /CBLK/ Y,J,A(10)
```

appear in two different subprograms, the translator converts them into Pascal as follows:

```
CBLK : record
          case CBLKINTE : boolean of
          true: (X : real;
                 I : integer;
                 A : array [1..10] of real);
          false: (Y : real;
                  J : integer;
                  A000001 : array [1..10] of real)
       end
```

Note that now the two array identifiers are unique within the record.

It is possible, however, in FORTRAN to use the same name in different subprograms to represent different variables in a COMMON block.

eg. In one subprogram

```
COMMON /CBLK/ X,I,A(10)
```

and in another subprogram

```
COMMON /CBLK/ A,I,B(10)
```

In cases such as this, the name of the duplicate variable encountered has its name altered by the translator to prevent duplication. The name is altered by adding an integer so that the name will be unique within the record description. This process satisfies the requirements of section 4.1.6.

Thus the above example is translated into the following Pascal record:

```
CBLK : record
        case CBLKINTE : boolean of
        true: (X : real;
               I : integer;
               A : array [1..10] of real);
        false: (A000001 : real;
                I000001 : integer;
                B : array [1..10] of real)
       end
```

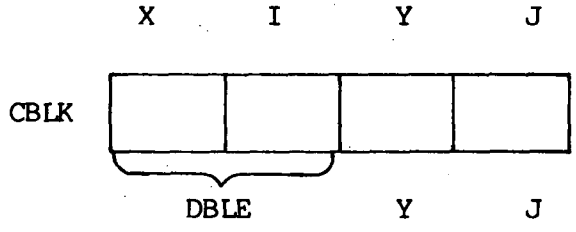
Variant records are also used when the storage sizes of the corresponding variables differ. In FORTRAN, it is possible to declare that a double precision (or complex) variable and a real (or integer

or boolean) variable share the same storage in a COMMON block. Rather than leave a storage unit unused the next variable in the block occupies the second storage unit of the double precision (or complex) variable.

eg. Consider the subprograms with COMMON statements

```
COMMON /CBLK/ X,I,Y,J
COMMON /CBLK/ DBLE,Y,J
```

The COMMON block can be pictured as:



Where the types of the corresponding elements differ the translator generates a variant record - one variant part for each different COMMON block declaration.

Thus, the above example will become

```
CBLK : record
         case CBLKINTE : boolean of
         true: (X : real;
                I : integer;
                Y : real;
                J : integer);
         false: (DBLE : double;
                 Y000001 : real;
                 J000001 : integer)
       end
```

The translator options, DOUBLE and COMPLEX (see 4.8), do not affect the declaration of COMMON records as these variable types take the same amount of storage, regardless of the way the options are set.

The FORTRAN EQUIVALENCE statement may be used in conjunction with COMMON statements to allocate the same storage to variables. Section 4.5.2.4 gives details of the translation of the EQUIVALENCE statement.

If the declaration of a COMMON block is identical in each subprogram then it may be possible to declare the entries of that COMMON block as variables in the Pascal outer block and not as a record. This approach would eliminate the need for using the record notation of Pascal as each variable is unique in the subprograms in which it is used. This approach is not taken because it destroys the consistency of the translation and it is not a general translation.

It is possible to surround the body of each Pascal procedure by a with statement which specifies each COMMON block record used in that procedure. This approach offers a degree of protection to the program and the COMMON block and it eliminates the need for prefixing each COMMON block variable by the COMMON block name in Pascal. However, the use of the with statement causes statements, which have no corresponding FORTRAN statement, to be generated and it may add a degree of difficulty to any subsequent editing. This approach is not taken by the translator.

The transfer of all COMMON blocks to Pascal records may result in a loss of efficiency when the Pascal program is compiled using

some compilers. For example, in B6700 Pascal records are stored as arrays and each access to that record causes the array to be indexed. In CDC Pascal records are stored in a manner similar to variables and a record access takes approximately the same time as does a variable access.

4.5.2.4 EQUIVALENCE Statement

An EQUIVALENCE statement in FORTRAN is of the form

$$\text{EQUIVALENCE } (k_1),(k_2), \ldots (k_n)$$

where each k is a list of the form

$$a_1,a_2, \ldots ,a_m$$

and each a is a variable name or array element name which contains only constants or subscripts. The EQUIVALENCE statement is used to permit the sharing of storage by two or more entities. Each element of the list is assigned the same storage (or part of the same storage) by the processor.

The translator converts all of those variables listed in an EQUIVALENCE statement list into a Pascal variant record. In B6700 Pascal, and most other versions of Pascal, variant records of the same size occupy the same storage area [Sale 1978b].

The names EQUIV01, EQUIV02, ... etc. are used for the names of the Pascal records and EQUIVI01, etc are used for the name of the tag field variable. The tag field variable is declared to be of type boolean if there are two elements in the EQUIVALENCE list. The labels used are then true and false. If there are more than two elements, the variable is declared to be of type integer, with the integers

1,2,3, ... etc. being used for the variant record labels.

If an identifier in the EQUIVALENCE statement is also used in a COMMON statement then the Pascal variant record becomes part of the record associated with the COMMON block.

When a FORTRAN identifier, used in an EQUIVALENCE statement, is later used in the Pascal program, it is preceded by the variant record name, eg. EQUIV01.A etc.

eg. (1) FORTRAN

```
EQUIVALENCE (A,B)
```

Pascal

```
EQUIV01 : record
   case EQUIVI01 : boolean of
   true: (A : real);
   false: (B : real)
end;
```

(2) FORTRAN

```
DIMENSION C(10)
EQUIVALENCE (A,C(3),D)
```

Pascal

```
EQUIV01 : record
   case EQUIVI01 : integer of
   1: (FILLER1 : array[0..1] of real;
       A: real);
   2: (C : array[1..10] of real);
   3: (FILLER2 : array[0..1] of real;
       D : real)
end
```

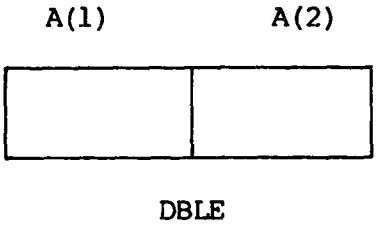
```
(3)  DIMENSION B(10)          CBLK : record
     COMMON /CBLK/ C(10)           case CBLKINTE : boolean of
     EQUIVALENCE (B(3),C(10))      true: (FILLER1 : array[0..6] of
                                                    real;
                                        B : array [1..10] of real);
                                   false: (C : array[1..10] of real)
                                end
```

The FORTRAN types DOUBLE PRECISION and COMPLEX take two storage units each for each variable and the other types (INTEGER, REAL and LOGICAL) take one storage unit. The translator options, DOUBLE and COMPLEX (see 4.8), do not affect the relationship of storage size between the different types. The algorithm used by the translator takes note of the type of each unit encountered and allocates it to the position the FORTRAN program intended. Filler elements are allocated by the translator to place an element in its correct position in a Pascal record. These elements are of type real if they occupy one storage unit or are an array [0..(n-1)] of real if they occupy more than one storage unit. The translator uses the names FILLER01, FILLER02, ...etc. for identifiers for the filler elements (see 4.1.6).

In FORTRAN, two real variables may share the same storage with a double precision variable - the second real variable occupies the second storage unit of the double precision variable.

eg.
```
DIMENSION A(2)
DOUBLE PRECISION DBLE
EQUIVALENCE (DBLE,A(1))
```

The storage map can be represented by:



The translator converts this FORTRAN case to:

```
EQUIV01 : record
   case EQUIVI01 : boolean of
   true: (A : array [1..2] of real);
   false: (DBLE : double)
end
```

4.5.2.5 EXTERNAL Statement

In FORTRAN, an EXTERNAL statement is of the form

$$\text{EXTERNAL } v_1, v_2, \ldots, v_n$$

where each v is an external procedure name.

There is no equivalent statement in Pascal. Pascal procedures and functions are declared in the main program and there is no facility for declaring them as external.

During the translation process, all procedures and functions referenced by a program must be present so that they may be declared in the main program, and so there is no need in Pascal for a statement similar to the EXTERNAL statement in FORTRAN.

4.5.2.6 Type Statements

In FORTRAN, a type statement is of the form

$$t\ v_1, v_2, \ldots v_n$$

where t is INTEGER, REAL, DOUBLE PRECISION, COMPLEX or LOGICAL and each v is a variable name, an array name, a function name or an array declarator. A type statement is used to override or confirm the implicit typing, to declare entities to be of double precision, complex or logical type, and it may supply dimension information.

In Pascal, the variable declaration takes the form

$$v_1, v_2, \ldots v_n : t$$

where

1) $v_1, \ldots v_n$ represent simple variables and t is a simple type (integer, real or boolean) or a defined type (eg. complex or double).

2) $v_1, \ldots v_n$ represent arrays of the same type and size and t takes the form

<u>array</u> [<bounds>] <u>of</u> <simple type>

A function may not be declared in this manner in Pascal. It must be declared with the body of the function under the <u>function</u> heading.

In Pascal it is not possible to mix the array declarators and simple variables in one statement. Rather, items of different type or size must be declared in separate declarations.

eg. The FORTRAN statement

```
INTEGER A,B,C(10),D(5)
```

becomes in Pascal

```
A,B : integer;
C : array [1..10] of integer;
D : array [1..5] of integer;
```

4.5.2.7 DATA Statement

In FORTRAN, a data initialization statement is of the form

$$\text{DATA } k_1/d_1/, \ldots k_n/d_n/$$

where each k is a list of variables or array elements and each d is a list of constants which may be preceded by a repeat count of the form J* where J is an integer constant. A data initialization statement is used to define initial values of variables or array elements.

There is no equivalent statement in Pascal and so the translator converts a FORTRAN DATA statement into assignment statements at the beginning of the executable part of the Pascal subprogram. This process adds a small overhead to the execution of the Pascal program.

In FORTRAN, Hollerith constants may appear in the list of constants to be assigned to a variable but this form of assignment is not allowed in Pascal and, when it occurs, an error message is printed.

When the repeat count is used to predefine an array, a for statement is used in Pascal to initialize the array. In all other cases, a simple assignment statement suffices.

eg. The FORTRAN statement

```
DIMENSION A(3)
DATA A/3*0.0/,B/1.0/
```

becomes in Pascal

```
for <temp. var.>:=1 to 3 do A[<temp. var.>] := 0.0;
B := 1.0;
```

Where a temporary variable is needed in a Pascal for statement, the name used is of the form DATATE01 ("DATA TEmporary", see 4.1.6).

4.5.2.8 FORMAT Statement

FORMAT statements are used in FORTRAN in conjunction with the input/output of formatted records to provide conversion and editing information between the internal representation and the external character strings.

A FORMAT statement is of the form

$$\text{FORMAT } (q_1 t_1 z_1 t_2 \ldots z_{n-1} t_n q_2)$$

where each q is a series of slashes or is empty

each t is a field descriptor or a group of field descriptors

each z is a field separator

n may by zero.

A FORMAT statement must be labelled.

In standard Pascal there is no equivalent statement and no means of performing formatted input/output. In B6700 Pascal formatted input/output is permitted and the translator uses these facilities to translate FORTRAN FORMAT statements into Pascal.

FORMATs in B6700 Pascal are declared separately at the beginning of a subprogram under the heading FORMAT. Each format declaration is identified by a unique name and the translator uses the following system of naming FORMATs in Pascal:

FORMAT<nn>

where <nn> is the FORMAT label in FORTRAN. This system preserves the

association between the FORTRAN and Pascal statements and satisfies the requirements of 4.1.6.

The format field descriptors in B6700 Pascal are based on those of B6700 Algol, which, in turn, were derived from FORTRAN. The format descriptors available in B6700 Pascal include many which are not available in standard FORTRAN but all those available in FORTRAN can be converted to equivalent descriptors in Pascal.

The FORTRAN format field descriptors and their corresponding B6700 Pascal format editing commands are summarized in Table 4.5

FORTRAN	Pascal
srFw.d	rFw.d
srEw.d	rEw.d
srGw.d	rRw.d
srDw.d	rEw.d or rDw.d
rIw	rIw
rLw	rLw
rAw	rCw
$nHh_1h_2...h_n$	"$h_1h_2...h_n$"
nX	Xn

Table 4.5 Format Field Descriptors

where

(1) s is an optional scale factor designator and takes the form nP where n, the scale factor, is an integer constant. The scale factor affects any following format conversion of F, E, D or G fields by multiplying that field by 10**n. However, the scale factor has no effect if there is an exponent in the external

field. In B6700 Pascal the scale factor format (the S format) is only associated with the R format and the multiplication is performed regardless of the external field. For these reasons, FORTRAN scale factors cannot be translated to B6700 Pascal and an error message is printed where they occur.

(2) r is the repeat count and indicates the number of times to repeat the succeeding basic field descriptor.

(3) w and n are non-zero integer constants representing the width of the field in the external character string.

(4) d is an integer constant representing the number of digits in the fractional part of the external character string.

(5) each h is one of the characters capable of representation by the processor.

The integer format field designator, I, converts directly to its B6700 Pascal equivalent. The real format field designators F and E also convert directly to their B6700 Pascal equivalents. If the double precision translator option, DOUBLE is false, all D format field designators are converted to E format field designators in B6700 Pascal, otherwise they remain unaltered.

The G format field designator has no direct equivalent in B6700 Pascal. The R format field designator corresponds when the external field follows an Ew.d format but in all other cases the FORTRAN designator is left justified with trailing spaces whereas the Pascal designator is right justified with leading spaces. As this designator is not used very heavily, this translation is performed and a warning message is printed where it occurs.

The logical format field descriptor, L, converts directly to its B6700 Pascal equivalent, but on output the effect is slightly different. Output from a FORTRAN variable using an Lw specification consists of (w-1) blanks followed by T or F for true or false respectively. In B6700 Pascal, as many characters from the words TRUE and FALSE as fit into the field width are used, left justified, blank filled. A warning message is printed by the translator to indicate a difference when the field width is greater than or equal to 2.

The character handling descriptor, A, is not translatable directly into Pascal. The C editing command corresponds most closely to FORTRAN's A descriptor but there are a few discrepancies. On input, if the field width is less than the number of characters capable of being held in a storage unit, then the FORTRAN standard specifies that the field will be held left justified, blank filled. Pascal's C editing command fills with zeros. Similarly on output, the fields are left justified and blank filled whereas the FORTRAN standard requires the field to be right justified with leading blanks. A warning message is printed by the translator on input, when the field width is less than the number of characters per word and on output when blank filling is required.

The Hollerith string descriptor, H, is converted directly to a Pascal string in accordance with 4.2.4. However, in FORTRAN, information may be read into a Hollerith string. This is not allowed in Pascal and an error message is printed where it occurs.

The blank field descriptor, X, converts directly to Pascal but

its syntax is reversed to Xn in B6700 Pascal.

In FORTRAN, the first character of each output record determines the vertical spacing to use with formatted output. When a FORMAT statement is used in association with a WRITE statement the first character of each output record is dropped. This presents no problem when there is only one record per FORMAT statement and that statement is only used with WRITE statements. If the FORMAT statement is used with both READ and WRITE statements, the translator creates two Pascal format statements called IFORMAT<nn> and OFORMAT<nn> for use with READ and WRITE statements respectively.

4.6 Procedures and Subprograms

4.6.1 Statement Functions

In FORTRAN, a statement function is defined internally to the program unit in which it is referenced. It is defined by a single statement similar in form to an arithmetic or logical assignment statement. All statement function definitions must precede the first executable statement and must follow any specification statements in the program unit.

A statement function definition takes the form

$$f(a_1,a_2, \ldots a_n) = e$$

where

f is the function name

e is an expression and the relationship between f and e must conform to the assignment rules of 4.5.1.1

the a's are the dummy arguments of the function whose names

merely indicate the type, number and order of the arguments.

A statement function is referenced by using its reference as a primary in an arithmetic or logical expression. Execution of a statement function reference results in the association of actual argument values with the corresponding dummy arguments in the expression of the function definition and an evaluation of the expression. The resultant value is then made available to the expression that contained the function reference.

In Pascal the concept of a statement function does not exist and the translator converts a FORTRAN statement function into a Pascal function definition.

Each of the dummy arguments in the FORTRAN statement function merely indicates the type and position of the actual argument. The function definition in Pascal has a similar system for indicating its argument ordering but each argument type must be explicitly declared in the function definition. Pascal function definitions must also indicate whether each argument represents a value parameter or a variable parameter. FORTRAN statement functions do not permit arguments to be altered during the evaluation of the function except perhaps if the expression references a function subprogram using a function argument as one of its arguments.

In general, therefore, arguments can be called by value but, to be safe, in the second case, a var parameter is used.

If the type of the function is COMPLEX or DOUBLE PRECISION and the translator option COMPLEX or DOUBLE is false (see 4.8.2 and

4.8.4) the function is not translated into Pascal, because Pascal requires its functions to be of a simple type. An error message is printed.

eg. (1) The FORTRAN statement function

```
FUNC1(A,B,C) = A + B - C
```

becomes in Pascal

```
function FUNC1(A,B,C : real) : real;
begin
   FUNC1 := A + B - C;
end
```

(2) The FORTRAN statement function

```
FUNC2(A,I) = A + FUNC3(I)
```

where FUNC3 is a function subprogram

becomes in Pascal

```
function FUNC2(A : real; var I : integer) : real;
begin
   FUNC2 := A + FUNC3(I);
end
```

4.6.2 Intrinsic Functions

Intrinsic functions in FORTRAN are predefined functions and have a special meaning and type if their names are not altered by redefinition. An intrinsic function is referenced by using its reference as a primary in an arithmetic or logical expression.

FORTRAN defines a total of 31 intrinsic functions many of which have no direct equivalent in Pascal. Table 4.6 summarises the FORTRAN

intrinsic functions and outlines equivalent Pascal expressions.

FORTRAN Intrinsic Function	No. args	Argument Type	Function Type	Meaning	Pascal Equivalent Expression or Function
ABS	1	real	real	|a|	abs
IABS		integer	integer		abs
DABS		double	double		see note on DOUBLE
AINT	1	real	real	sign of a times	trunc
INT		real	integer	largest integer	trunc
IDINT		double	integer	<=|a|	see note on DOUBLE
AMOD	2	real	real	a_1 mod	see note on mod
MOD		integer	integer	a_2	a_1 mod a_2
AMAX0	>=2	integer	real	max(a_1,	max(a_1,a_2,...) *
AMAX1		real	real	a_2,...)	max(a_1,a_2,...)
MAX0		integer	integer		max(a_1,a_2,...)
MAX1		real	integer		trunc(max(a_1,a_2,..))
DMAX1		double	double		see note on DOUBLE
AMIN0	>=2	integer	real	min(a_1,	min(a_1,a_2,...) *
AMIN1		real	real	a_2,...)	min(a_1,a_2,...)
MIN0		integer	integer		min(a_1,a_2,...)
MIN1		real	integer		trunc(min(a_1,a_2,..))
DMIN1		double	double		see note on DOUBLE
FLOAT	1	integer	real	integer to real conversion.	ignored as integers and reals are compatible in Pascal
IFIX	1	real	integer	real to integer conversion	trunc
SIGN	2	real	real	sign of	see note

FORTRAN Intrinsic Function	No. args	Argument Type	Function Type	Meaning	Pascal Equivalent Expression or Function
ISIGN		integer	integer	$a_2 * \|a_1\|$	see note
DSIGN		double	double		see note
DIM	2	real	real	a_1-min(	(a_1-min(a_1,a_2))
IDIM		integer	integer	a_1,a_2)	(a_1-min(a_1,a_2))
SNGL	1	double	real	convert double to real	see note on DOUBLE
REAL	1	complex	real	obtain real part of complex argument	A.RE
AIMAG	1	complex	real	obtain complex part of complex argument	A.IM
DBLE	1	real	double	convert real to double	see note on DOUBLE
CMPLX	2	real	complex	$a_1+a_2\sqrt{-1}$	A.RE := A1 A.IM := A2 see note on COMPLEX
CONJG	1	complex	complex	obtain conjugate of complex argument	A.RE := A1.RE A.IM := -A1.IM

Table 4.6 Intrinsic Functions

Note that the Pascal functions min and max are not standard Pascal but they are available in B6700 Pascal and are useful extensions to Pascal even though they have a variable number of parameters and their calling mechanism is therefore non-standard. The function type is the same as the type of the arguments.

The functions AMOD, SIGN, ISIGN and DSIGN have no equivalent functions or expressions in Pascal. The mod operator in Pascal requires two integer arguments. However, each of these functions may be coded as a Pascal function and, where required, it is declared in the Pascal outer block by the translator. The equivalent functions are

```
function AMOD (A1,A2 : real) : real;
begin
   AMOD := A1 - trunc(A1 / A2) * A2;
end;

function SIGN (A1,A2 : real) : real;
begin
   if (A2 < 0) then
      SIGN := -abs(A1)
   else
      SIGN := abs(A1);
end;

function ISIGN (A1,A2 : integer) : integer;
begin
   if (A2 < 0) then
      ISIGN := -abs(A1)
   else
      ISIGN := abs(A1);
end;
```

```
function DSIGN(A1,A2 : double) : double;
begin
   if (A2 < 0) then
      DSIGN := -abs(A1)
   else
      DSIGN := abs(A1);
end;
```

Each of these functions in FORTRAN is not defined when the second parameter is equal to zero. The above definition for AMOD will produce an undefined result (division by zero) when the second parameter is zero but the definitions for the SIGN functions will produce the same result as A2 being positive. This action is compatible with the undefined approach taken by FORTRAN.

If the translator option, DOUBLE, is set to true, then an assumption is made that all of the double precision FORTRAN functions except DSIGN can be converted to Pascal in a manner similar to that for real functions.

That is, DABS is converted to abs and it is assumed that abs handles double precision variables. Similarly IDINT is converted to trunc, DMAX1 to max and DMIN1 to min. It is assumed that a function sngl exists to convert the FORTRAN function SNGL.

The FORTRAN function DBLE is treated as an extension to the way in which FLOAT is treated and not converted to a Pascal function. It is assumed that double and real types are compatible in the manner in which integers and reals are compatible.

If the translator option DOUBLE is set false then no double precision functions are assumed to exist in Pascal. All double precision variables are then converted to records consisting of two real parts and all processing is done on the first real variable. All double precision functions are converted to the equivalent functions for real variables. Thus DABS is considered as ABS, IDINT as INT, DMAX1 as MAX1, DMIN1 as MIN1 and DSIGN as SIGN. The functions SNGL and DBLE are ignored during translation as they then involve two real parameters.

If the translator option COMPLEX is set true then the FORTRAN functions involving complex types are assumed to have equivalent functions in Pascal. It is assumed that the function cmplx exists in Pascal and is equivalent to the FORTRAN function CMPLX. The Pascal function conjg is assumed to be equivalent to the FORTRAN function CONJG. In each case, arguments and results are assumed to be equivalent.

If the translator option COMPLEX is false then complex variables are stored as records and equivalent Pascal statements are given in Table 4.6. No complex functions are assumed to exist in this form of Pascal.

4.6.3 External Functions

An external function in FORTRAN is defined externally to the program unit which references it. It is headed by a FUNCTION statement and called a function subprogram.

A FUNCTION statement is of the form

t FUNCTION f (a_1,a_2, ... a_n)

where

(1) t is the function type or is empty

(2) f is the symbolic name of the function to be defined

(3) the a's, called dummy arguments, are each a variable name, an array name, or an external procedure name.

An external function is referenced by using its reference as a primary in an arithmetic or logical expression. The actual arguments must agree in order, number and type with the corresponding dummy arguments. The function may define or redefine any of its arguments to return results in addition to the value of the function.

The translator declares all arguments, in Pascal, as value arguments, except where the argument is defined, redefined or used in a call to another subprogram, in which case it is defined as a variable parameter.

The function statement in FORTRAN is converted to its Pascal equivalent

function f (<var>a_1:<type>; <var>a_2:<type>; ...) : t

In FORTRAN, a function-value-variable may be used in expressions as a variable. However, in Pascal, if the function-value-variable is used in an expression recursion will be assumed by the compiler. As this is not allowed in standard FORTRAN, a new variable will be generated by the translator and used as the function-value-variable throughout the subprogram and its value is assigned to the function-value-variable immediately before a RETURN statement is executed. The new variable is generated only when the

function-value-variable is used in an expression.

In Pascal a function type must be a scalar, pointer or subrange type. A structured type is not permitted. If a user defined function is of type COMPLEX, or of type DOUBLE PRECISION and the translator option DOUBLE is set false, it is not translated to Pascal but an error message is printed.

The FORTRAN standard defines a set of basic external functions. These and their Pascal equivalent functions or expressions are summarised in Table 4.7.

FORTRAN Basic External Function	No. Arg.	Argument Type	Function Type	Definition	Pascal Equivalent Function or Expression
EXP	1	real	real	e**a	exp
DEXP	1	double	double		see note on DOUBLE
CEXP	1	complex	complex		see note on COMPLEX
ALOG	1	real	real	log(a)	ln
DLOG	1	double	double		see note on DOUBLE
CLOG	1	complex	complex		see note on COMPLEX
ALOG10	1	real	real	$\log_{10}(a)$	ln(a)/2.3025850930
DLOG10	1	double	double		see note on DOUBLE
SIN	1	real	real	sin(a)	sin
DSIN	1	double	double		see note on DOUBLE
CSIN	1	complex	complex		see note on COMPLEX
COS	1	real	real	cos(a)	cos
DCOS	1	double	double		see note on DOUBLE
CCOS	1	complex	complex		see note on COMPLEX
TANH	1	real	real	tanh(a)	tanh *
SQRT	1	real	real	sqrt(a)	sqrt
DSQRT	1	double	double		see note on DOUBLE
CSQRT	1	complex	complex		see note on COMPLEX
ATAN	1	real	real	arctan(a)	arctan
DATAN	1	double	double		see note on DOUBLE
ATAN2	2	real	real	arctan (a_1/a_2)	arctan2 *
DATAN2	2	double	double		see note on DOUBLE
DMOD	2	double	double	$a_1 (\text{mod } a_2)$	see note on DOUBLE
CABS	1	complex	real	modulus	see note on COMPLEX

Table 4.7 External Functions

If the double precision translator option, DOUBLE, is true, then pre-defined functions are assumed to exist in Pascal to handle the corresponding FORTRAN functions. DEXP is converted to dexp in Pascal, DLOG to dln, DLOG10 to dln(A)/2.3025850930, DSIN to dsin, DCOS to dcos, DSQRT to dsqrt, DATAN to darctan and DATAN2 to darctan2. A function dmod, similar to amod (see 4.6.2), is created in the Pascal outer block when DMOD is used in the FORTRAN program.

If the double precision option, DOUBLE, is false, then the double precision type in FORTRAN is considered as a Pascal record of two reals, only one of which is used in expressions. The double precision functions then have no equivalents in Pascal but each function is converted as the corresponding real function is converted. Thus DEXP is considered as EXP, DLOG as ALOG, DLOG10 as ALOG10, DSIN as SIN, DCOS as COS, DSQRT as SQRT, DATAN as ATAN, DATAN2 as ATAN2, and DMOD as AMOD.

If the translator option, COMPLEX, is true, then it is assumed that the complex functions cexp, clog, csin, ccos, csqrt and cabs exist in Pascal and are equivalent to the FORTRAN functions CEXP, CLOG, CSIN, CCOS, CSQRT and CABS respectively.

If this option is set false, then each complex variable in FORTRAN is converted to a Pascal record containing two real variables. No complex functions are assumed to exist in Pascal. An expression involving complex variables in FORTRAN is converted to two equivalent expressions in Pascal - one for each part of the complex variable. Complex functions are considered as their real counterparts

- CEXP as EXP, CLOG as ALOG, CSIN as SIN, CCOS as COS and CSQRT as SQRT - for conversion purposes. The function CABS is converted to the Pascal expression

sqrt(A.RE*A.RE + A.IM*A.IM)

The functions marked * in the table are not standard Pascal functions but they are available in B6700 Pascal. They are useful extensions to standard Pascal and necessary for the translation of FORTRAN programs to a neat and efficient form in Pascal.

4.6.4 SUBROUTINE Subprograms

An external subroutine in FORTRAN is defined externally to the subprogram which references it. An external subroutine is defined by FORTRAN statements headed by a SUBROUTINE statement and is called a subroutine subprogram. A SUBRCUTINE statement is of one of the forms:

SUBROUTINE s (a_1,a_2, ... a_n)

or SUBROUTINE s

where

(1) s is the symbolic name of the subroutine

(2) the a's, called dummy arguments, are each a variable name, an array name, or an external procedure name.

In Pascal, a FORTRAN subroutine statement corresponds to a procedure statement which takes the form

procedure s (<var>a_1 : type; ....)

In each language, the subprogram may define or redefine one or more of its arguments so as to effectively return results. In FORTRAN

the symbolic name of the subroutine may not appear in any statement in that subroutine except the SUBROUTINE statement itself. Hence, recursion is not possible as it is in Pascal.

A subroutine is referenced by a CALL statement (see 4.5.1.8). The actual arguments must agree in order, number and type with the corresponding dummy arguments in the defining program.

FORTRAN permits the use of Hollerith constants as actual arguments and this is an exception to the rule requiring agreement of type. Pascal does not permit this form of disagreement of types and when it occurs, a warning message is printed by the translator.

All FORTRAN arguments are translated into Pascal value arguments except those which are external procedure names, or those which are defined, redefined or used in subprogram calls in the subprogram, in which case var arguments are used.

4.6.5 BLOCK DATA Subprograms

In FORTRAN, a BLOCK DATA statement is of the form

BLOCK DATA

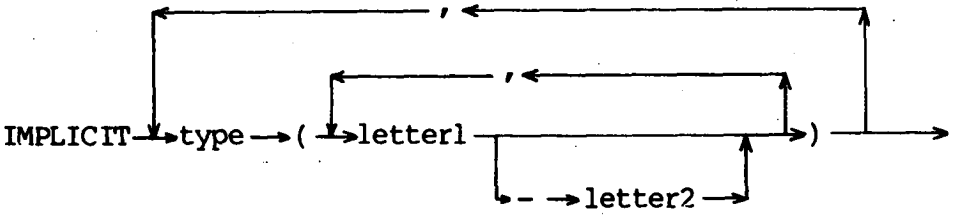
This statement may only appear as the initial statement of specification subprograms which are used to enter initial values into elements of labelled common blocks.

The translator treats BLOCK DATA statements in a manner similar to DATA statements (see 4.5.2.7) and places the resulting assignment statements at the beginning of the executable part of the main Pascal program.

4.7 Non-Standard Features

4.7.1 IMPLICIT Statement

This statement is accepted by the translator because it is available in most versions of FORTRAN, is a useful extension to FORTRAN and is easy to implement. Most manufacturers have implemented this statement using the syntax shown in the following railroad diagram:



where

(1) type is a FORTRAN data type

(2) letter1 and letter2 are alphabetic characters with letter2 following letter1 in the alphabet.

The IMPLICIT statement is used in FORTRAN to override or confirm the default implicit declaration specifications. The statement specifies that all implicitly declared identifiers beginning with letters between letter1 and letter2 inclusive are to be declared as type "type".

The translator declares all implicitly defined variables and functions in the subprogram in which the IMPLICIT statement occurs, in accordance with the IMPLICIT statement.

4.8 Translator Options

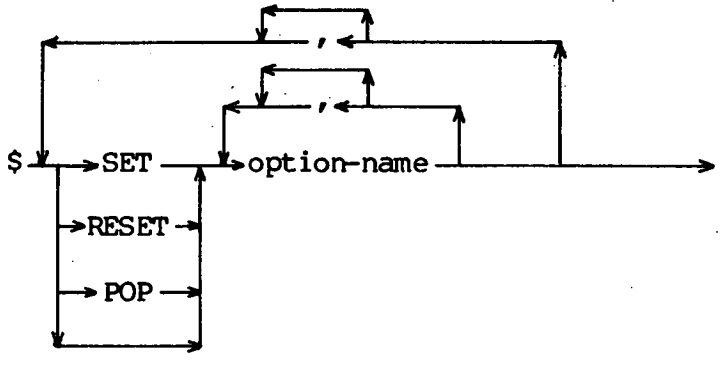
Translator options are specified on special translator option

records and allow the user to control some of the functions of the translation, such as requesting a listing of the source text, or the generation of an output disk file, or specifying the way in which COMPLEX is handled.

A translator option record is recognised by the translator since it has a $ in the first character position of the record, or a space followed by a $ in the first two character positions of the input record.

Most of the options have a boolean value associated with them and it may be SET or RESET. Each boolean option has a stack (limited to a maximum of 48) of values associated with it.

The syntax of a compiler option record is:



NOTE: this syntax only handles boolean valued options.

The appearance of an option-name after SET forces the stack to be pushed down by one and the top-of-stack value to be SET. RESET functions similarly except that the top-of-stack value is RESET. POP causes the top-of-stack value to be discarded, and the stack 'popped': all elements moved up by one.

This concept of translator options allows a user to change the

value of an option throughout the translation. The process was originally written by the author in conjunction with B6700 Pascal and taken from B6700 Algol.

The options accepted by the translator are shown in Table 4.8.

Boolean-valued	Numeric-valued	Other
$	ERRORLIMIT	INCLUDE
B6700	IDLENGTH	PAGE
CDC		
COMPLEX		
DISK		
DOUBLE		
ERRLIST		
FLIST		
FORSTMT		
ICL		
INCLNEW		
LISTINCL		
MERGE		
NEW		
OMIT		
PLIST		
SEQ		
STANDARD		
WARNINGS		
user-options		

Table 4.8 Translator Options

The following discussion pertains to those options which control the type of Pascal to be produced by the translator. A discussion of all other options may be found in the B6700 Pascal Reference Manual or any other Burroughs language reference manual.

4.8.1 B6700/CDC/ICL/STANDARD (default=all reset)

These options control the type of Pascal to be produced by the translator. When one of these options is SET the version of Pascal to be produced is tailored to that machine (or standard).

Only one of these options may be SET at any one time - an attempt to SET another one of the options will result in the initial option being RESET. The options IDLENGTH, COMPLEX and DOUBLE are also controlled by setting the above options. COMPLEX and DOUBLE are RESET when any of these options is SET and the option IDLENGTH is set to

72 if B6700 is SET
10 " CDC " "
8 " ICL " "
8 " STANDARD " "

4.8.2 COMPLEX (default=reset)

If this option is SET, the translator assumes that the version of Pascal to be produced handles the data type COMPLEX. The data type is assumed to be an ordered pair of real numbers and the syntax for handling this data type is assumed to be similar to that of FORTRAN. Standard functions (see 4.6.2) for handling basic COMPLEX operations are assumed to exist.

If this option is RESET, the version of Pascal to be produced is assumed to have no facilities for handling a COMPLEX data type. When a COMPLEX data type occurs in a FORTRAN subprogram, the translator converts it into a Pascal record (see 4.2.3).

4.8.3 DISK (default=reset)

If this option is SET, the translator will produce a disk file containing the Pascal program. This file may then be edited or used as input to a Pascal compiler. The contents of the file are the same as the Pascal listing produced (except for the heading and trailer information).

If the option is RESET, no disk file is produced.

4.8.4 DOUBLE (default=reset)

This option is handled in a manner similar to that of COMPLEX (see 4.8.2).

If the option is SET, the translator assumes that the version of Pascal to be produced handles the data type DOUBLE PRECISION. The data type in Pascal is assumed to have the same relationship with the type REAL as is required by the FORTRAN standard. Standard functions (see 4.6.2) for handling basic DOUBLE PRECISION operations are assumed to exist in the version of Pascal.

If the option is RESET, the version of Pascal to be produced is assumed to have no facilities for handling a double precision data type. When a DOUBLE PRECISION data type occurs in a FORTRAN subprogram, the translator converts it to a Pascal record (see 4.2.2) consisting of two real parts.

4.8.5 FLIST/PLIST (default=both set)

These options control the listing of the FORTRAN subprograms

(FLIST) and the Pascal program produced (PLIST). Both options are SET initially but may be SET or RESET at any point during the translation process. They operate in a manner similar to LIST in B6700 Pascal.

4.8.6 FORSTMT (default=reset)

When this option is SET, the user guarantees that all FORTRAN DO statements with an incrementation value of 1 have an initial value less than or equal to the final value and the control variable is not used in an expression after the loop has been exited. The translator then converts all such statements into a Pascal for statement.

When the option is RESET, a construct involving a repeat ... until loop is used for this sort of DO loop (see 4.5.1.13).

4.8.7 IDLENGTH (Special numeric default)

IDLENGTH is not a boolean option and it cannot be SET or RESET. It is associated with a numeric value which is the maximum length of identifiers produced by the translator.

The syntax for setting this option is shown below. The default value is 8, and the option cannot be set to a value less than 8.

5. SPECIFIC IMPROVEMENTS

5.1 Structure

Transformations that are performed on programs are usually done to minimise some cost associated with the program. The cost might be the amount of storage used for data, the number of instructions in the program or the number of instructions executed while performing the program. A program may be transformed into an equivalent one for reasons which are external to the program itself, such as making the program easier to debug, or matching it to the system under which it is to be executed.

The results of [Bohm & Jacopini 1966] state that any proper computer program can be constructed from a series of the basic building blocks discussed in section 2.3.3. Bohm and Jacopini conjectured that auxiliary variables are necessary, in general, to convert an unstructured program to a structured one, according to the rules of section 2.3.3. [Ashcroft & Manna 1971] and [Peterson, Kasami & Tokura 1973] supported this conjecture. If one is willing to accept the use of extra state-variables or control flags, or the introduction of some redundant coding, then a successful transformation can be contemplated.

Dijkstra, however, stated in [Dijkstra 1968a]

> "The exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one."

In other words, if automatic goto elimination procedures are applied

to badly structured programs, the resulting programs can be expected to be badly structured too. However, not all programs are badly structured nor do they have arbitrary flow diagrams; most programs are reasonably well structured. One purpose of the translator is to highlight the structure of a program - not alter it. This is done by the use of the structured statements in Pascal and by adopting a layout of the program statements which leads to a clearer program.

5.1.1 Techniques of Structuring Programs

There are three common techniques for converting unstructured programs to structured ones : duplication of coding; the state-variable approach; and the boolean flag approach. Each technique will be discussed below.

(i) Duplication of Coding

This technique involves duplicating the coding of each block of code which does not correspond to the "one entry, one exit" philosophy of structured programming. For example, if we take the unstructured program segment shown in Figure 5.1 and apply the technique of duplication of coding, we obtain the program segment shown in Figure 5.2 (Example due to [Yourdon 1975]).

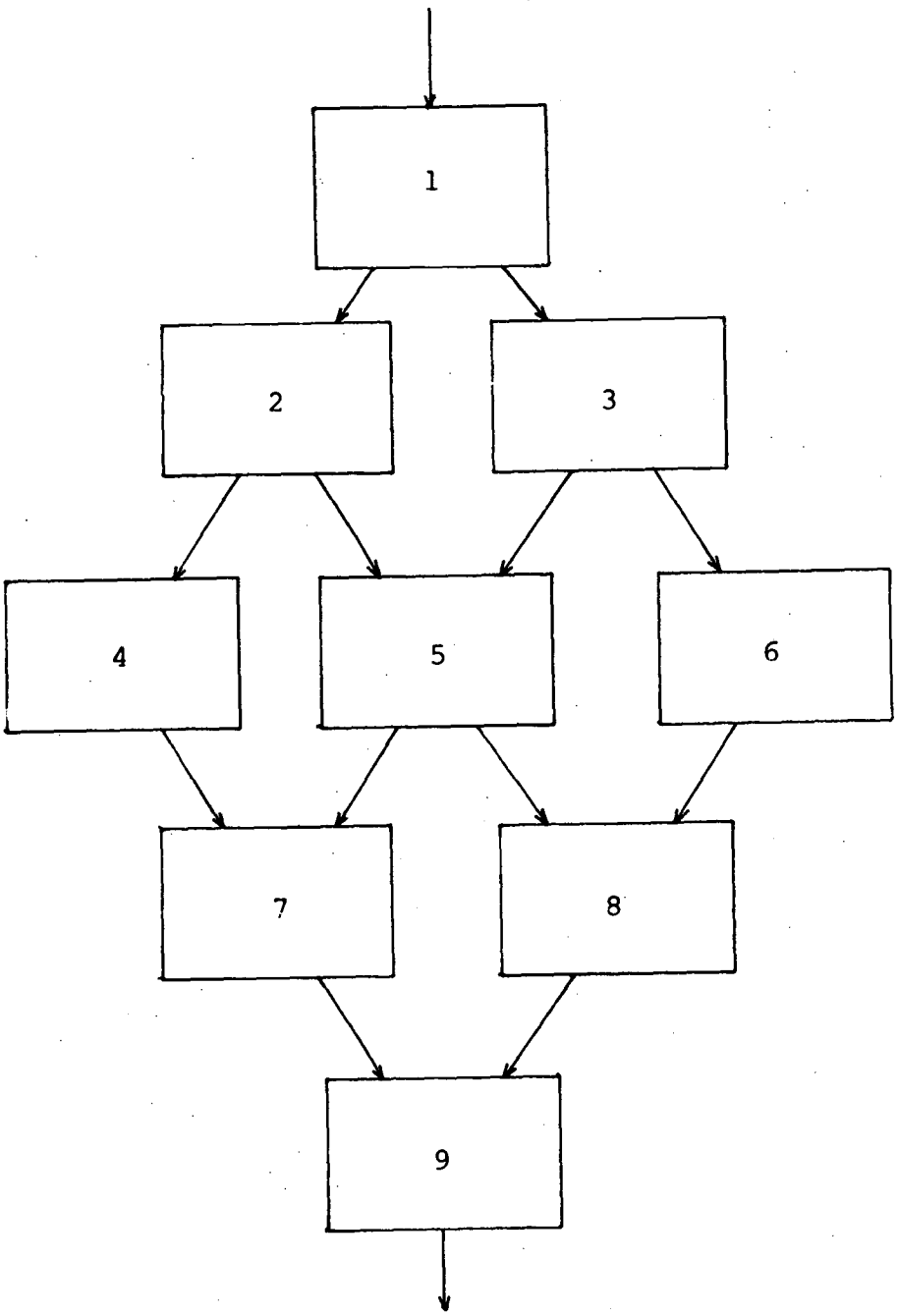


Figure 5.1 An Unstructured Program

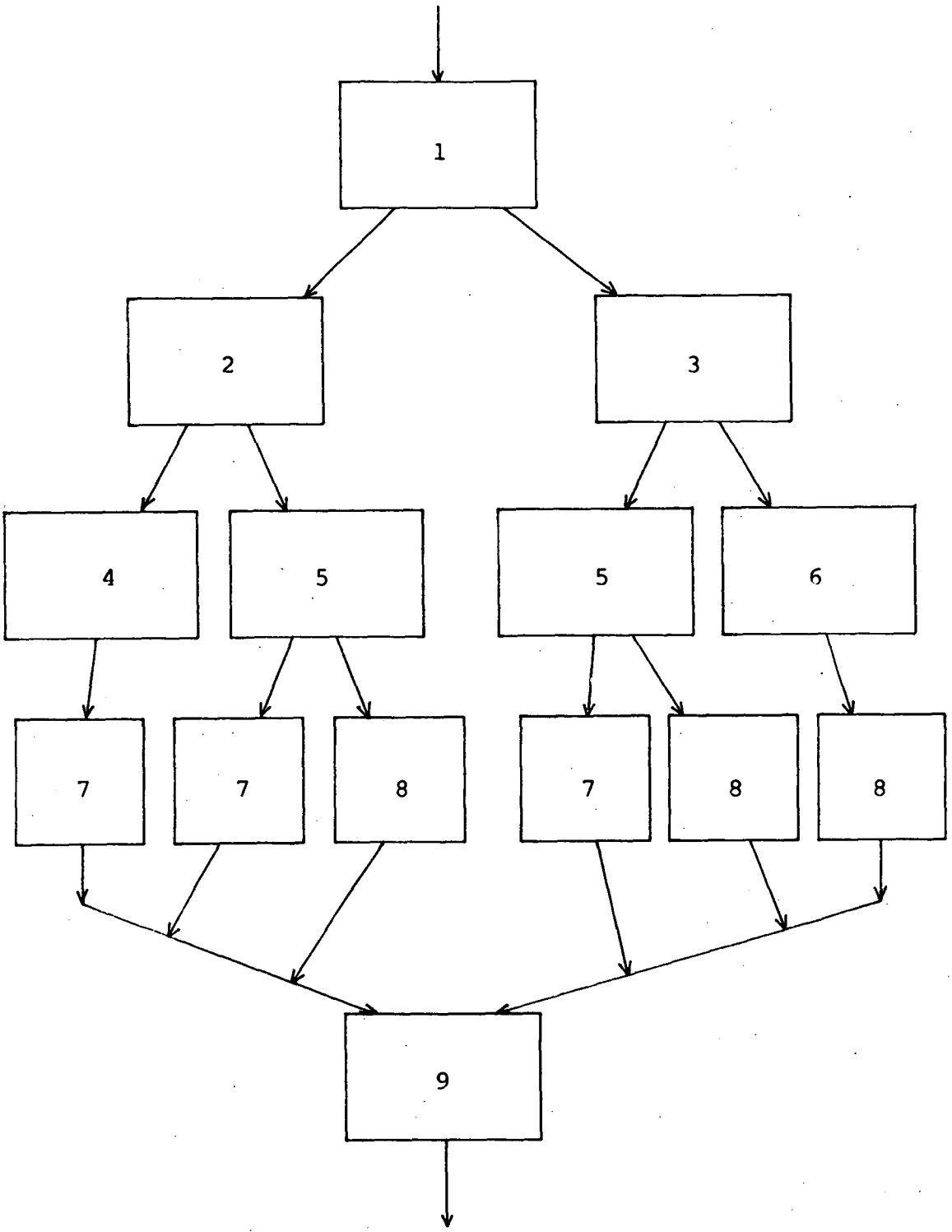


Figure 5.2 A Redesigned Form of Figure 5.1

One of the strongest arguments in favour of this technique is that the original program can be broken down to a simple IF ... THEN ... ELSE ... structure. The thinking and design process involved is considerably less complex than the original lattice structure and therefore less prone to errors.

This technique can be applied generally to any program which does not have loops. Yourdon [Yourdon 1975] claims that this technique will not work, in general, for program segments with loops but only for network or lattice structures. The two techniques to be discussed below deal with loop structures.

The obvious disadvantage of this technique is that it requires more memory than the original approach. If the code to be duplicated involved a few statements then the cost involved would be worth while. If the module to be duplicated contained a substantial amount of coding then a callable subroutine could provide an answer. Such an approach should produce a formal subroutine with formal parameters so that its correctness could be determined without regard to the context in which it executes. If this approach is taken, we produce multiple calls on a single copy of a subroutine - an approach which involves a relatively small amount of overhead.

This technique does not involve an increase in the number of possible paths through the program segment but it does involve an increase in the number of blocks in the segment. The technique can be applied to loop structures under certain conditions - the most suitable structures being loops of the form "two entries, one exit". Such a case is shown in Figure 5.3.

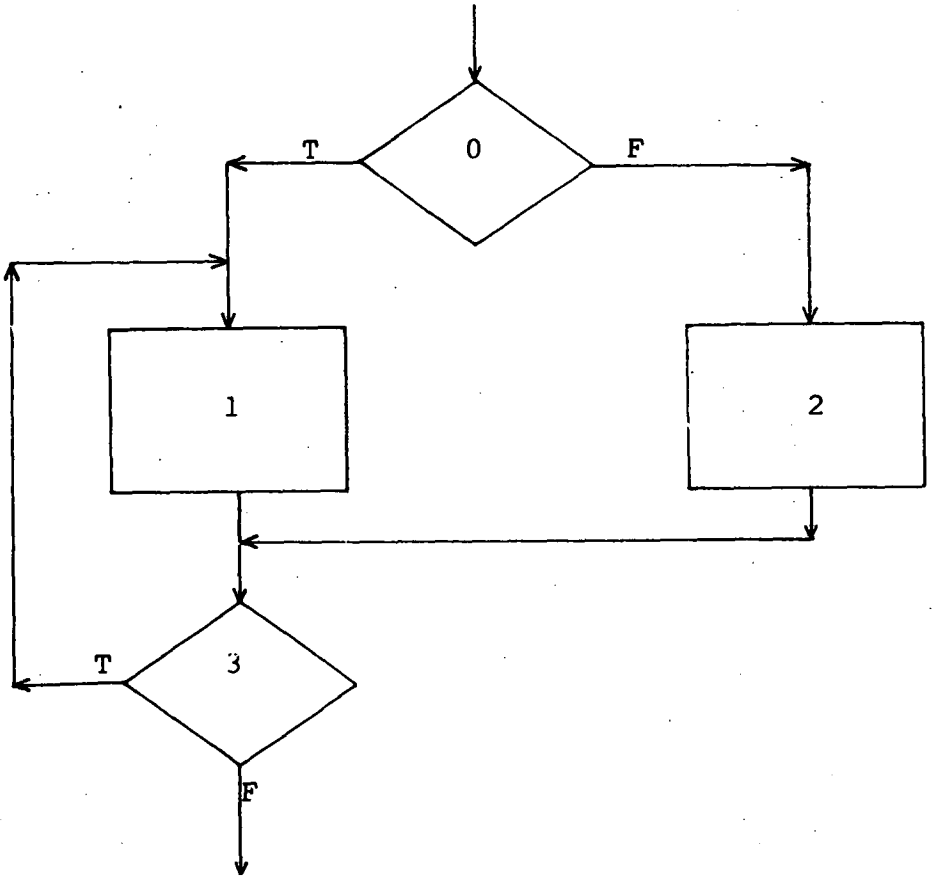


Figure 5.3 A "Two Entry, One Exit" Loop

In this case, block 1 is duplicated and a while loop constructed from blocks 1 and 3. A solution of the form

```
if pred(0) then
    1
else
    2;
while pred(3) do
    1;
```

satisfies the requirements of structured programming. The application of this technique to loop structures often involves the "unwinding" of a loop, i.e. converting a repeat loop into a while loop or vice versa with either additional code or extra conditions being applied.

(ii) The State-Variable Approach

Another technique for converting unstructured programs to structured ones was suggested in a paper by Ashcroft and Manna [Ashcroft & Manna 1971]. The technique is a slight variation on the approach of Bohm and Jacopini [Bohm & Jacopini 1966] and involves the addition of a single integer variable which serves as a 'program counter' or state-variable. Each block of the unstructured program is given a number and the blocks are replaced in the structured program by blocks which perform the same function but also set the integer variable to the integer which identifies the successor block in the original program. The decision boxes are converted in a similar fashion with the auxiliary variable being set to the integer value representing the block to which control passes after the decision has been made. The flow of control is then implicit in the sequence of values of the state-variable.

By using this approach, any program can be represented by a flowchart of the form shown in Figure 5.4.

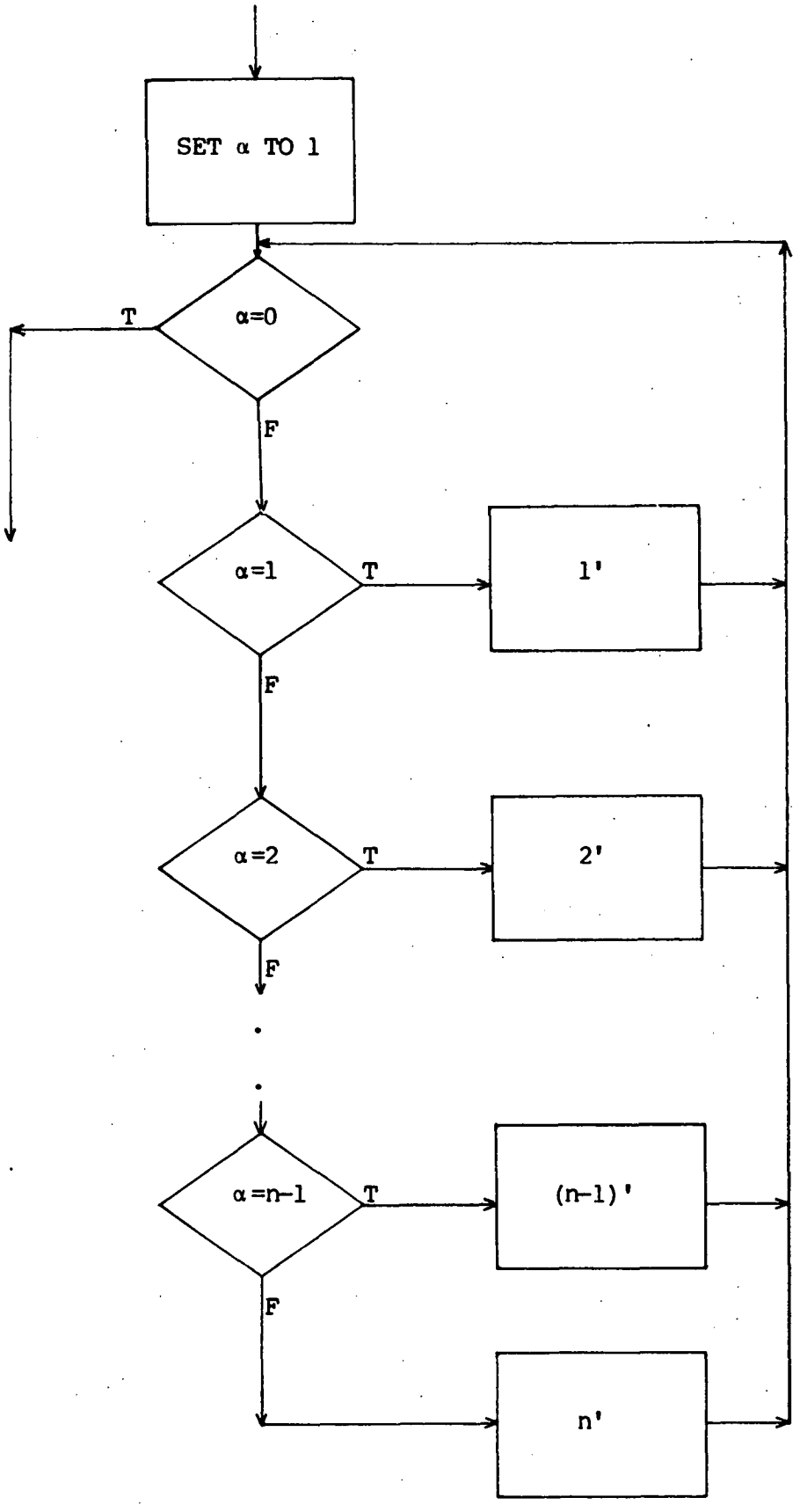


Figure 5.4 The Translated Form using the State Variable Approach

This form of conversion has the advantage that it can be applied to any program and in a mechanical fashion with relative ease. The flow diagram produced is always of the same form as that shown in Figure 5.4 and the diagram can be increased indefinitely without increasing the complexity of the approach. It is relatively simple to trace the state-variable during the execution of a program of this form and this should give a programmer a reasonably clear idea of the flow of control through the program for that execution and a debugging aid.

However, constructions such as the one shown in Figure 5.4 are undesirable not only because of their inefficiency, but because they destroy the topology of the original structure and thus make a source listing extremely difficult to understand. The flow of control is easy to implement but it is hard to trace actions through the program and for this reason it is an undesirable method to use.

The number of possible paths through a flow diagram of this form is infinite and a correct execution of a program of this form depends upon the setting of the control variable. The construction serves to illustrate the point that adding a control variable is an effective device for eliminating the goto.

(iii) The Boolean Flag Technique

Ashcroft and Manna [Ashcroft & Manna 1971] have given algorithms for translating arbitrary programs into goto-less form (with the addition of variables) which preserves some of the topology of the original program. This technique is typically applied to loops and it

involves the initialization of the flag before the loop is entered; the control of execution of the loop until the flag is set appropriately; and finally some condition inside the loop determining when the flag should be set. The technique may also be applied to network or lattice structures and then it involves the initialization of the flag before a decision block; the setting of the flag instead of executing a block of common code; the testing of the flag after the paths from the decision block merge and the execution of the common block if the flag has been set. This process is illustrated in Figure 5.5 and the figure shows an optimization whereby the null path indicator is set after the decision block.

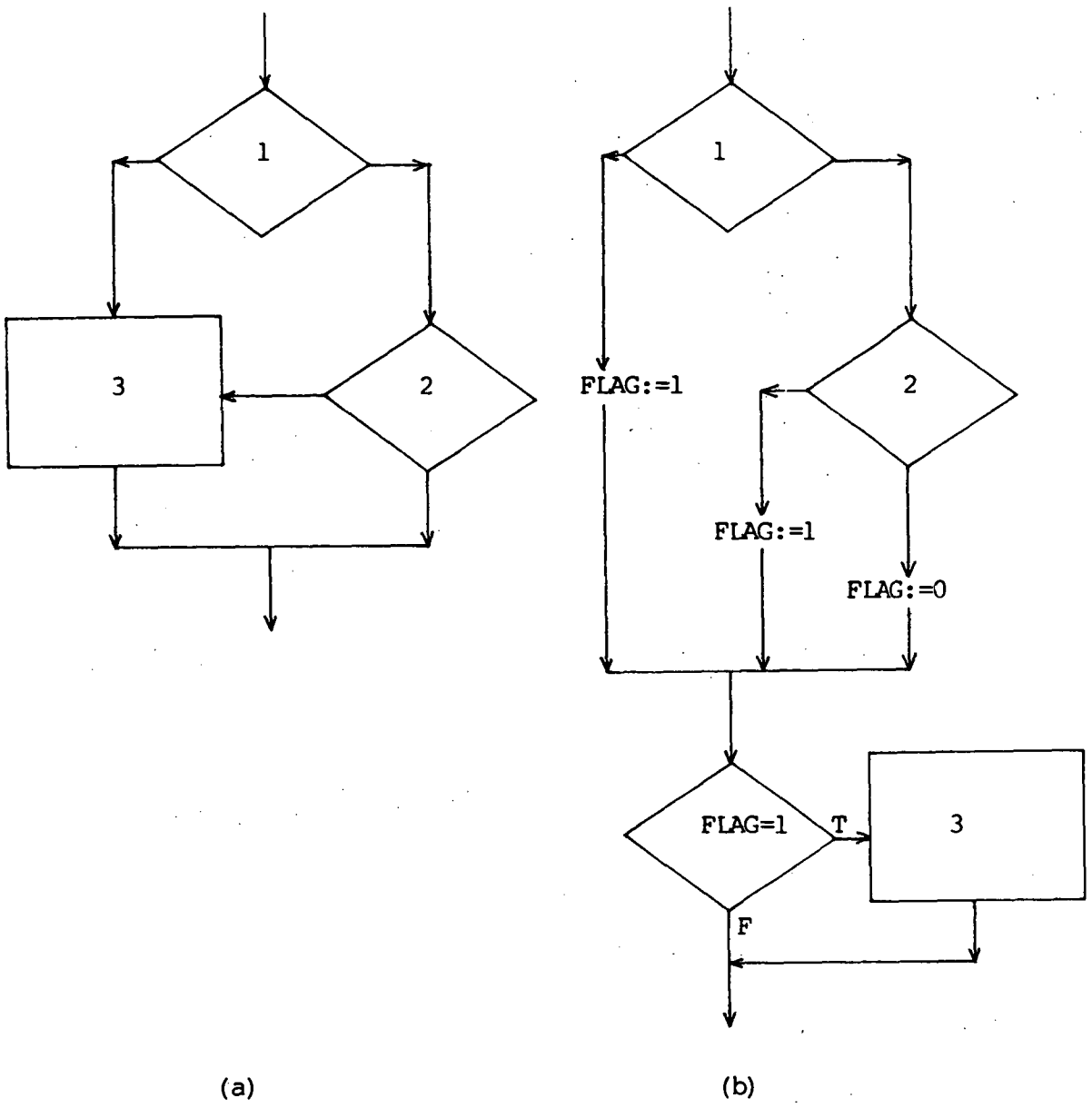


Figure 5.5 The Application of the Boolean Flag Technique to a Lattice Structure

This process involves a small overhead of an extra variable and code to set and test that variable. Yourdon [Yourdon 1975] claims that it is not always necessary to add an extra variable as the nature of the program sometimes permits decisions to be joined together by a boolean operator, or a program variable might contain

enough information to control the program execution. He also gives an example which combines the boolean variable technique and the duplication of coding technique to produce a structured program, and this approach combines the benefits of each technique and nullifies some of the drawbacks.

The number of possible paths through a flow diagram of the form shown in Figure 5.5 increases by a factor of 2 after the application of the boolean flag technique. By correctly setting the boolean variable the number of paths is reduced to the number in the unstructured program segment.

5.1.2 Structuring used by the Translator

The method of structuring used by the translator is based on three main criteria:

(i) the generality of the technique;

(ii) the style of programs produced by the technique;

(iii) the number of paths produced by the technique for a program segment.

The duplication of coding technique is not generally applicable to program segments containing loops and therefore this technique is not suitable as a general solution. The other two techniques are applicable to programs with and without loops.

The style of program produced by adopting the use of a state-variable is always of the same form (as shown in Figure 5.4). This style is disliked because it destroys the form of the original program and it makes a program listing extremely difficult to

follow.

The style of program produced by using the boolean variable technique bears some resemblance to the original program. A comparison of the diagrams in Figure 5.5 shows that block 3 has been replaced in the unstructured diagram by statements which set the boolean variable. Block 3 has been pushed past the merge point of the paths originating from blocks 1 and 2 and the execution of block 3 is controlled by the setting of the boolean variable. The new flow diagram is equivalent to the original one and each path through the original flow diagram corresponds to a path through the new flow diagram.

In general, in lattice structures, blocks with more than one entry are replaced by a statement which sets an auxiliary variable and the blocks from that point to the merge point of the paths entering the block are removed to a point after the merge point and their execution is conditional upon the setting of the auxiliary variable. This action produces an equivalent flow diagram whose form corresponds to the requirements of structured programming. The null path, or a path which bypasses the multiple entry block, does not set the auxiliary variable and therefore the new code placed after the merge point is not executed.

A comparison of Figures 6.4 and 6.5 shows that the form of the diagrams is similar - the major difference being that the path from block 2 to block 5 in the unstructured diagram has been replaced by a path from block 2 to the head of the loop (block 1) and an auxiliary variable is set along this path. The boolean condition in block 1 has

been altered and the execution of blocks 4 and 5 is controlled by the auxiliary variable.

The application of the boolean variable technique to a loop with many exits involves altering all paths, except one, out of the loop so that the paths set an auxiliary variable and transfer control to the terminal point of the loop. The terminal condition is altered so that the loop is exited if the auxiliary variable has been set, and a conditional statement is inserted after the loop termination to transfer control to the appropriate block depending on the value of the auxiliary variable. This action forces all paths out of a loop to leave the loop at one point, and then control is transferred to the various succeeding blocks. This action produces a well structured flow diagram equivalent to the original unstructured diagram.

The number of paths through a structure varies markedly with the application of each technique. Figure 5.6 gives a summary of the number of paths produced by each technique when the number in the unstructured flow diagram was n.

duplication of coding	state-variable	boolean flag
n	infinite	2n if the structure was a lattice n if the structure contained a loop

Figure 5.6 Summary of Paths Produced by Each Technique

In the cases of the state-variable and the boolean flag techniques the actual number of potential paths reduces to the original number (n) by setting the auxiliary variable correctly.

The translator uses the boolean flag technique to restructure programs because the technique is applicable generally; the style of program produced by it bears some resemblance to the original program; it is relatively easy to follow a program listing; and the technique produces the smallest number of additional paths through a flow diagram.

Peterson et al. [Peterson, Kasami & Tokura 1973] give a boolean flag algorithm for converting a flowchart to a well-formed flowchart, and this algorithm has been used as the basis for the block ordering of the FORTRAN to Pascal translator. The major difference is that Peterson's concept of "do forever" and "exit" statements in the coding of the algorithm have been replaced by the repeat statement in Pascal. Simple cases (those with one exit at the first or last block)

are translated using Pascal's repeat or while statements and the loop is controlled by the condition specified in the FORTRAN program. All other loops are translated into Pascal repeat loops controlled by an auxiliary variable.

5.2 Subprogram Ordering

In FORTRAN, an executable program consists of a main program plus any number of subprograms, external procedures or both. No mention is made in the FORTRAN standard of any ordering of these program components and they can therefore occur in any order.

eg. Main program followed by all the subprograms

or

Some subprograms, followed by the main program, followed by further subprograms

etc.

In Pascal, procedures and functions are included in the scope of the procedure which references them, or global to the set of procedures which reference them. Each procedure, therefore, lies within the outer bounds of the main program. FORTRAN, on the other hand, treats each subprogram as a separate entity and no subprogram is permitted to lie within the bounds of another subprogram.

Some reordering of subprograms during the translation process may be necessary and the translator handles the ordering of subprograms in one of two ways:

(1) If no main program exists in the FORTRAN deck then the translator assumes a FORTRAN subprogram library is being

translated to Pascal. Not enough information about interprogram calls exists in a subprogram library to allow a reordering algorithm to be applied, and the translator converts the subprograms in the order of the original FORTRAN deck.

(2) If a main program exists in the FORTRAN input deck then a complete FORTRAN program is translated to a Pascal program. Error messages are printed for any missing subprograms or for any unreferenced subprogram.

The first pass of the translator creates a list of all external subprograms for each subprogram. An incidence matrix, A, is then constructed and Warshall's algorithm [Warshall 1962] is used to determine whether any recursive links exist between subroutines. Although recursion is not permitted in FORTRAN, two subprograms may mutually call each other - a process usually controlled by a flag to prevent recursion. Warshall's algorithm produces a reachability matrix, R, with each entry, r_{ij}, indicating whether or not a path exists from subprogram i to subprogram j [Harary 1969].

Commencing with the main program, the translator builds a list of subprograms to be declared inside the body of the current subprogram, i, in the following manner:

If the row, A_i, is not zero then

1. Determine the subprogram, j, with the maximum number of callers by taking the column sum of column j of the incidence matrix. If two or more subprograms have the same number of callers, then select the one reachable from more subprograms at this level.

2. If subprogram j is a member of a recursive loop (i.e. $r_{jj}=1$) then discover all the other members of the loop and if they have more than

one caller, declare them forward, and zero the entry, r_{jj}, in R.

3. Enter subprogram i into the list of subprograms to be called by the current subprogram.

4. Zero column j in both the incidence matrix and the reachability matrix.

Repeat steps 1 - 4 until row i is completely zero. Then, for each entry in the list of subprograms repeat the whole process.

Proof of Algorithm:

Warshall's algorithm determines all paths i to j for all subprograms. The ordering algorithm commences with the main program and continues until all links from a subprogram in the list have been eliminated.

Therefore, all paths in the program have been considered.

By selecting the program with the maximum number of callers in step 1, the 'most called' procedures are declared first. When a column is zeroed, that subprogram cannot be considered again by the algorithm and so each subprogram is entered into a list once.

Chapter 6 contains some examples of the procedure ordering.

5.3 Pascal Program Layout

One aspect of programming style which affects the usefulness of programs is their readability. Peterson [Peterson 1977] defines readability as the ability of a programmer to pick up a program, read it and understand it. Many aspects of style affect readability,

including variable names, commenting, modularity and formatting. The translator does not alter the FORTRAN variable names, comments or the modular structure of the FORTRAN program, but it does format the Pascal program produced.

Strictly speaking, formatting is a matter of personal taste. There is no proven best way of formatting a program although a number of authors [Ledgard et al. 1977, Peterson 1977, Sale 1978a and Mohilner 1978] have suggested various formatting styles.

The FORTRAN language permits programmers to use formatting in their program layout but in practice very few programmers do. Of all the FORTRAN programs tested by the translator only about 1% use any kind of consistent layout. The translator, therefore, cannot rely on its input program as a basis for laying out the resultant Pascal program.

Of all the styles suggested in the literature, the classical style is probably the most common. This style was used in the Algorithms section of the Communications of the ACM for Algol 60 and gained its popularity from there. As Sale [Sale 1978a] points out there are a number of problems with this style including paper wastage, editing inconvenience and understanding. The author prefers a personal style which he has developed during his programming experience and the style is very similar to that advocated by Sale.

The translator indents all declarations from their heading, and takes a new line for each new declaration which differs from the previous declaration.

```
eg. var
        A : array [1..10] of integer;
        I,J : integer;
```

In the main program body, the translator indents all internal statements of a compound statement by one level unless a predefined maximum indentation limit has been reached and then no further indenting occurs until the level of nesting has been reduced. All begins are placed on the same line as the statement to which they belong and an end associated with an else or until is placed on the same line as that else or until.

```
eg.  if condition then begin
        statements;
     end else begin
        statements;
     end;

     while condition do begin
        statements;
     end;

     repeat begin
        statements;
     end until condition;
```

The begin - end brackets are always used - even if only one statement is controlled because the pattern is then regular and the addition and deletion of statements at a later date can be made without altering the begin - end brackets.

This style is clear and it eliminates a number of problems found in the other styles. For example, the template for an _if_ includes _end else begin_ so no confusing errors due to _elses_ are possible. The inner statements of a compound statement are indented by one level, not two as is the case with the classical style. This means that more programs containing much structuring detail can be handled before problems associated with the right margin are encountered. The style is more compact vertically than other styles because each _begin_ is placed on the initial line of the statement, and if an _end_ is associated with a part of the statement construct (eg. _until_, _else_) that _end_ is placed on the same line as the statement construct.

The style consists of a series of line-oriented templates and each line is terminated by a semicolon except after a _begin_ (where the semicolon does not matter) or where a line has to be split over several lines. As a corollary, editing is simplified as an editing insertion or deletion involves only the line concerned.

6. FIELD TRIALS

6.1 Specific Test Cases

In this section, a number of examples are given and they illustrate how the translator handles certain unstructured FORTRAN programs, and how it reorders FORTRAN subprograms. Only a limited number of cases may be examined here; an extensive set of examples is given in Appendix B.

It should be emphasized again that the translator makes no attempt to optimize the resultant Pascal program. An inefficient FORTRAN program may lead to an inefficient Pascal program.

6.1.1 Lattice Structure with Abnormal Selection Paths

Figure 5.5(a) illustrates the graph of a program in which a code segment (3) is executed under different sets of conditions. It is an unstructured graph which must be changed into a structured form. The translator changes this graph into the form shown in Figure 5.5(b).

Figure 6.1 contains a FORTRAN subroutine which uses "GOTOs" to implement an abnormal selection path. The structured version, in Pascal and produced by the translator, is illustrated in Figure 6.2. Note that the piece of code whose execution is controlled by the boolean variable, is only one statement and it could be argued that code duplication would be more appropriate in this case. However, the example was chosen because it is concise and it illustrates the application of the boolean variable technique and a nested IF ... THEN ... ELSE ... structure.

```
      SUBROUTINE XMPLE1(M,N)
C        ABNORMAL SELECTION PATH
      IF(M.LT.0) GOTO10
      IF(M.GT.0) GOTO10
      N=0
      GOTO 20
   10 N=-1
   20 M=M+1
      RETURN
      END
```

Figure 6.1 An Unstructured FORTRAN Subroutine

```
PROCEDURE XMPLE1 (VAR M , N : INTEGER ) ;
VAR
(*        ABNORMAL SELECTION PATH                                    *)
   BOOLEA01 : BOOLEAN ;
BEGIN
   IF ( M < 0 ) THEN BEGIN
      BOOLEA01 := TRUE;
   END ELSE BEGIN
      IF ( M > 0 ) THEN BEGIN
         BOOLEA01 := TRUE;
      END ELSE BEGIN
         N := 0 ;
         BOOLEA01 := FALSE;
      END;
   END;
   IF BOOLEA01 THEN BEGIN
      N := - 1 ;
   END;
   M := M + 1 ;
END;
```

Figure 6.2 A Structured Pascal Version of Figure 6.1

Figure 6.2 illustrates the layout employed by the translator. In this case, the number of lines of program has grown from 10 in the FORTRAN subprogram to 23 in the Pascal procedure. The increase is due to three factors - the use of blank lines, the declaration of a variable and the stylistics of the layout. The use of begin - end

brackets, although only one inner statement is controlled, is a factor in the increase in the number of lines. Figure 6.3 shows the same Pascal program, with the begin - end brackets surrounding the single statement removed. The number of lines of Pascal has now been reduced to 17, but the regularity of the style has been compromised.

```
PROCEDURE XMPLE1 (VAR M , N : INTEGER ) ;
VAR
(*      ABNORMAL SELECTION PATH                                    *)
   BOOLEA01 : BOOLEAN;
BEGIN
   IF (M<0) THEN BOOLEA01 := TRUE
   ELSE IF (M>0) THEN BOOLEA01 := TRUE
        ELSE BEGIN
             N := 0;
             BOOLEA01 := FALSE;
        END;
   IF BOOLEA01 THEN N:= -1;
   M := M + 1;
END;
```

Figure 6.3 A Modified Version of Figure 6.2

6.1.2 Loop with a Single Entry and Multiple Exits

Figure 6.4 depicts the form of a single entry, two exit loop. To rewrite this in a structured manner the graph of the program must be altered and a new variable and predicate added to make the logically possible flows in the new graph execute the same structurally possible flows which occurred in the original graph. The manner in which this is automatically performed is indicated in Figure 6.5.

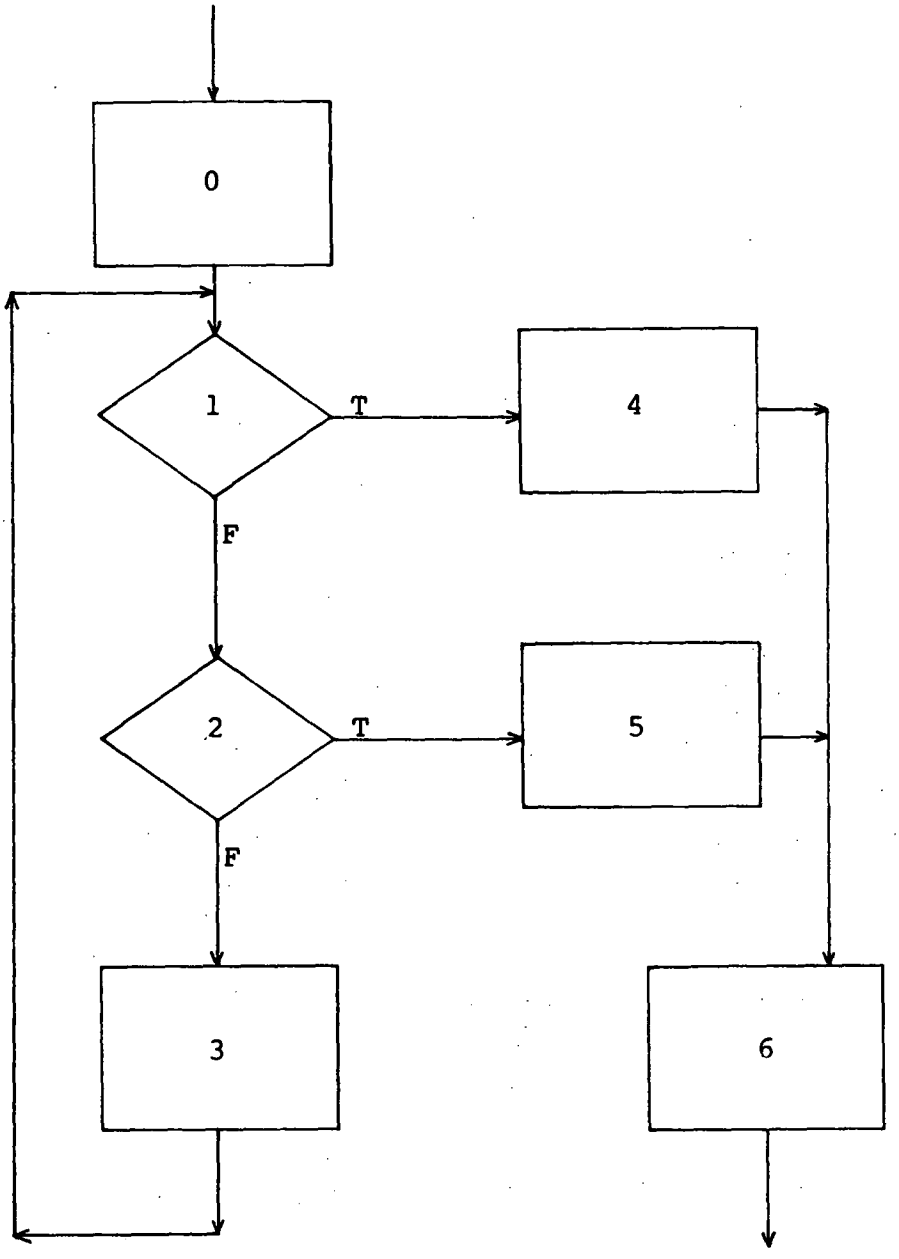


Figure 6.4 A One Entry, Two Exit Iteration

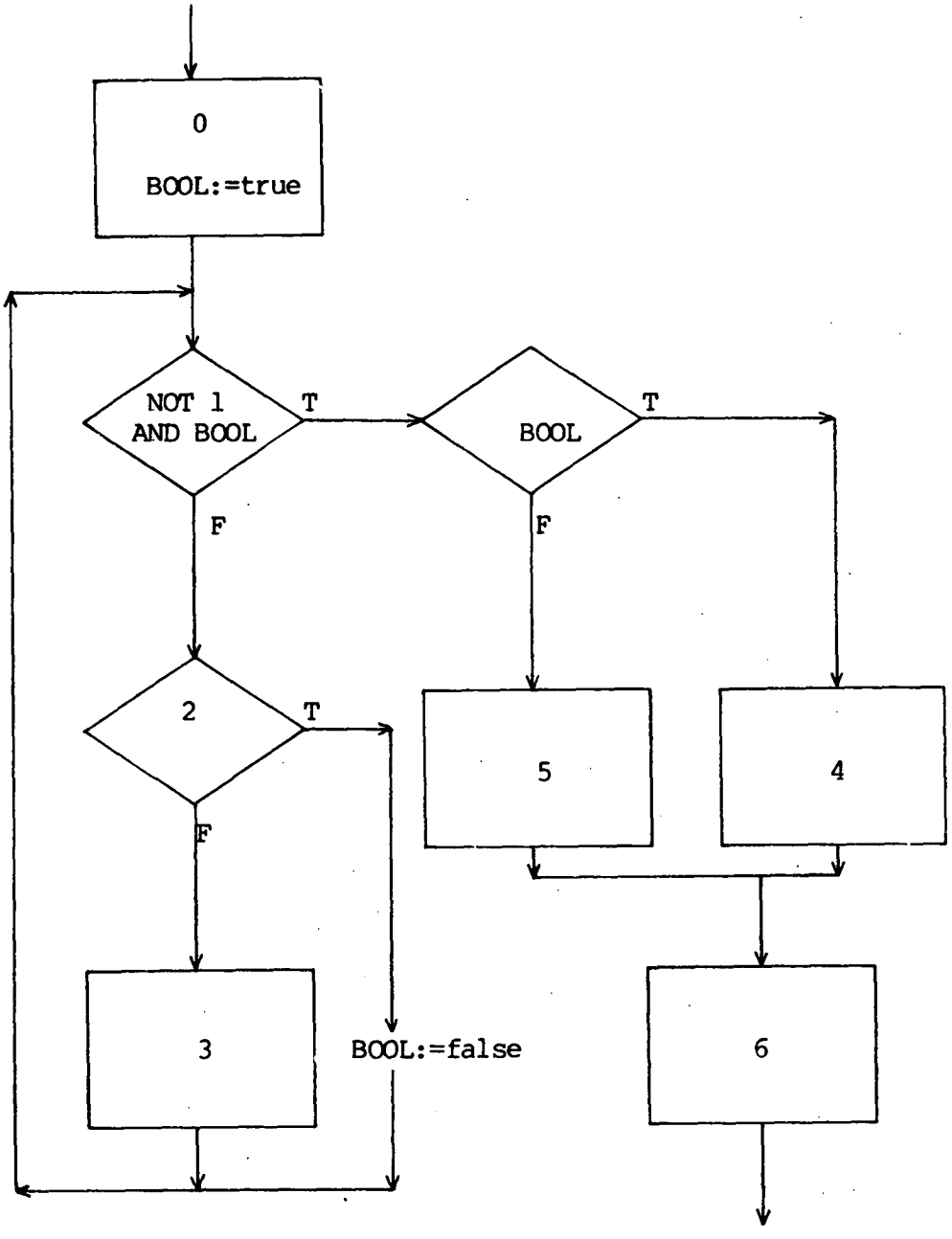


Figure 6.5 A Structured Version of Figure 6.4 with a Variable and Predicate Added

Two assignment statements, one new variable, a new predicate, and a modified predicate have been added to allow this program to be written in a structured manner. A FORTRAN program using this structure is given in Figure 6.6 and the translated version in Pascal

is shown in Figure 6.7.

```
      SUBROUTINE XMPLE2(L,IARRAY,N)
C       LOOP WITH MULTIPLE EXIT PATHS
      INTEGER IARRAY(10)
      I=10
   30 IF (I.LT.1) GOTO 10
      IF (L.EQ.IARRAY(I)) GOTO 20
      I=I-1
      GOTO 30
   10 N=0
      GOTO 40
   20 N=I
   40 CONTINUE
      RETURN
      END
```

Figure 6.6 A Single Entry, Two Exit Iteration in FORTRAN

```
PROCEDURE XMPLE2 (L : INTEGER ; IARRAY : IARRAYTY ; VAR N : INTEGER ) ;
VAR
(*       LOOP WITH MULTIPLE EXIT PATHS                                  *)
   I : INTEGER ;
   BOOLEA01 : BOOLEAN ;
BEGIN
   I := 10 ;
   BOOLEA01 := TRUE;
   WHILE (NOT ( I < 1 ) AND BOOLEA01 ) DO BEGIN
      IF ( L = IARRAY [ I ] ) THEN BEGIN
         BOOLEA01 := FALSE;
      END ELSE BEGIN
         I := I - 1 ;
      END;
   END ;
   IF BOOLEA01 THEN BEGIN
      N := 0 ;
   END ELSE BEGIN
      N := I ;
   END;
END;
```

Figure 6.7 The Automatically Rewritten Version of XMPLE2

If the number of exits from the loop exceeds two, the

translator declares an integer variable in the Pascal program, sets the variable to zero before entering the loop, tests the variable for zero in each iteration, sets the variable to a unique non-zero integer should the execution path leave the loop and, after exiting the loop, generates a Pascal case statement to handle the exit path taken. The following Pascal program segment illustrates the process:

```
CASEVAR := 0;
while ( not (cond.) and (CASEVAR=0)) do begin
   if (pred1) then begin
      CASEVAR := 1;
   end else begin
        .
        .
   end;   {of if}
end;   {of while}
case CASEVAR of   {which path was taken? }
0:          {normal exit}
     .
1:          {first abnormal exit}
     .
2:          {second abnormal exit}
     .
end;        {of case}
```

6.1.3 Simple Subroutine Linkage with COMMON Blocks

Figure 6.8 illustrates the calling linkages of a simple subroutine calling sequence. In the sequence, the main program calls two subroutines (SB and SC) and subroutine SC calls another subroutine, SD. The FORTRAN language does not permit subroutine nesting and it allows subprograms to appear in any order. One ordering is illustrated in Figure 6.9 but the order is immaterial to the translator. This program also illustrates the way in which the translator handles FORTRAN COMMON blocks. In this case, each subprogram uses a COMMON block and some of the variable names in the COMMON block differ from subprogram to subprogram.

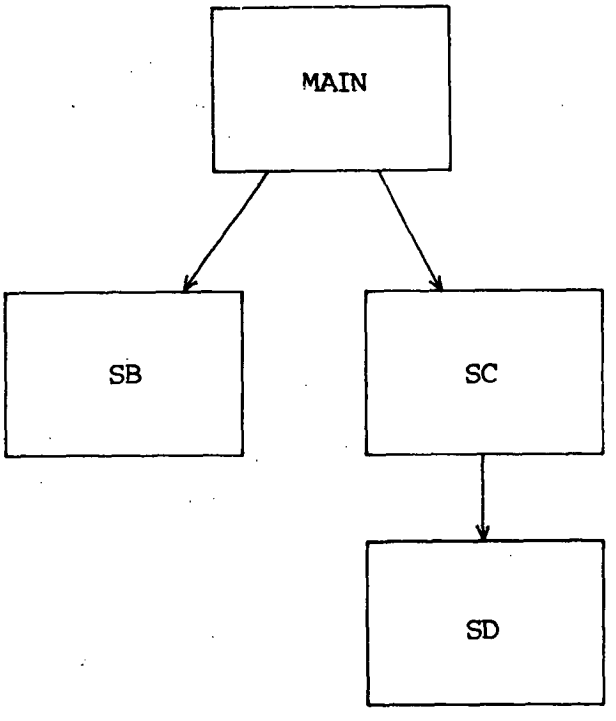


Figure 6.8 A Simple Subroutine Calling Sequence

```
      COMMON A(10),B,C
      CALL SB
      CALL SC
      STOP
      END
      SUBROUTINE SB
      COMMON A(10),X,C
      C=7
      RETURN
      END
      SUBROUTINE SC
      COMMON A(10),B,Y
      Y=5
      CALL SD
      RETURN
      END
      SUBROUTINE SD
      COMMON Z(10)
      A=8
      RETURN
      END
```

Figure 6.9 A FORTRAN Implementation of Figure 6.8

The translator recognises that subroutine SD is only called by subroutine SC and places SD within the scope of SC. Figure 6.10 illustrates the translator output for the FORTRAN program of Figure 6.9.

```
PROGRAM FORTPASC;
       (*********)

VAR

    BLANKCOM : RECORD
       CASE CASEC00C : BOOLEAN OF
       TRUE:
          (A : ARRAY [1 .. 10] OF REAL ;
          CASE CASEC01 : BOOLEAN OF
          TRUE:
             (B : REAL ;
             CASE CASEC002 : BOOLEAN OF
             TRUE:
                (C : REAL
                );
             FALSE:
                (Y : REAL
                )
             );
          FALSE:
             (X , CCCCCC02 : REAL
             )
          );
       FALSE:
          (Z : ARRAY [1 .. 10] OF REAL
          )
    END;

PROCEDURE SB ;

BEGIN
    BLANKCOM . CCCCCC02 := 3 ;
END;

PROCEDURE SC ;

PROCEDURE SD ;

VAR

    A , B : REAL ;

BEGIN
    A := B ;
END;


BEGIN
    BLANKCOM . Y := 5 ;
    SD ;
END;


BEGIN
    SB ;
    SC ;
END.
```

Figure 6.10 The Pascal Program Produced from the Program in Figure 6.9

In the COMMON block in each subprogram, different names are used for the COMMON variables and the translator generates a Pascal variant record for the COMMON block. The translator uses the name 'BLANKCOM' for the record because the FORTRAN program used blank COMMON and it allocates three new variables as tag variables in the variant records. In each case, the new name generated is unique because it is greater than six characters in length (the FORTRAN limit) and/or it contains a unique integer as part of the name.

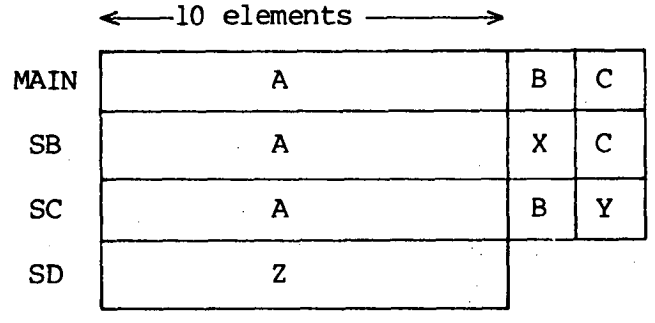


Figure 6.11 The COMMON Block Layout

Figure 6.11 illustrates the layout of the FORTRAN COMMON block in the program of Figure 6.9. The Pascal language only permits the latter part of a record to vary and the translator, therefore, must generate a unique name for element C of the COMMON block in subroutine SB. The translator uses a unique integer to generate the new name and forces the new name to be greater than six characters in length to overcome any possible uniqueness problems. All variable names are then prefixed by the record name when used in Pascal statements as this is required by the Pascal syntax.

The tag variables allocated by the translator are declared to be of type boolean if two variants exist at that level of the record,

and type integer if more than two variants are used. When the tag is of type integer the variant case-constants used are the integers 1,2,3, ...

6.1.4 Recursive Subroutine Linkage

Figure 6.12 illustrates a subroutine flow diagram in which a recursive loop (SB - SD - SE) is present. Although the FORTRAN language expressly forbids the use of recursion, two or more subroutines may form an apparently recursive structure provided that, during program execution, recursion does not occur. This situation is rare but, to be complete, the translator must cater for it. Figure 6.13 shows a FORTRAN implementation of this apparently recursive structure. The subroutine flow diagram is that shown in Figure 6.12, but during execution not all of the paths can be taken. When the logical variable FLOW is set .TRUE. the path SE - SB cannot be taken and no recursion takes place. Similarly, if FLOW were set .FALSE. the path SB - SD could not be taken and no recursive loop would be formed.

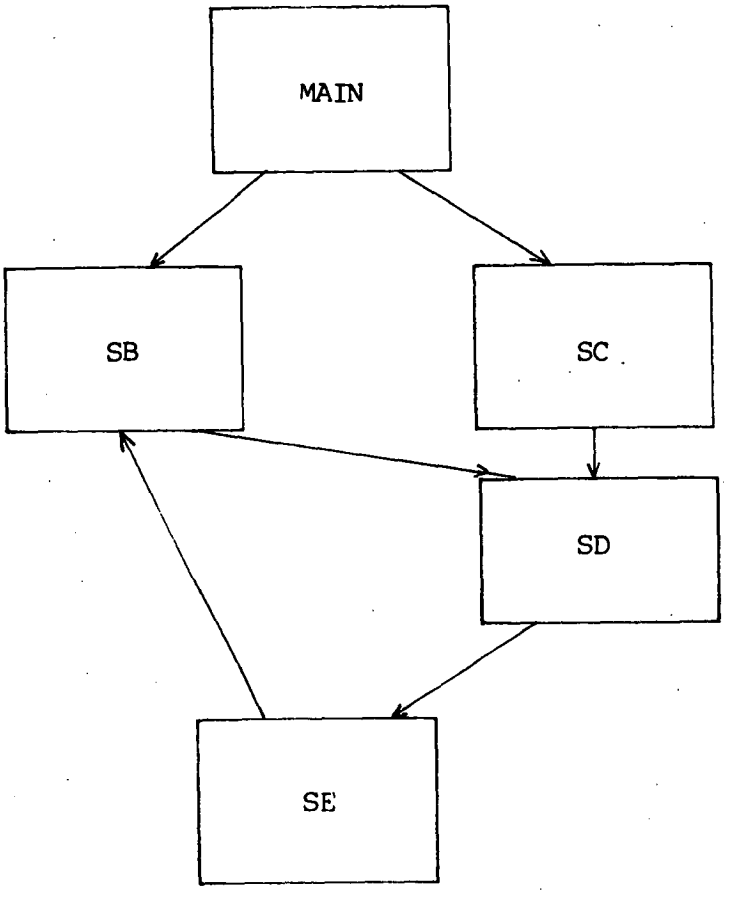


Figure 6.12 A Recursive Subroutine Structure

```
C
C
      LOGICAL FLOW
      COMMON /CONTRL/ FLOW
      FLOW = .TRUE.
      CALL SB
      CALL SC
      STOP
      END
C
      SUBROUTINE SB
      LOGICAL FLOW
      COMMON /CONTRL/ FLOW
      IF (FLOW) CALL SD
      RETURN
      END
C
      SUBROUTINE SC
      CALL SD
      RETURN
      END
C
      SUBROUTINE SD
      A=SE(3)
      RETURN
      END
C
      FUNCTION SE(B)
      LOGICAL FLOW
      COMMON /CONTRL/ FLOW
      IF (.NOT. FLOW) CALL SB
      SE=B
      RETURN
      END
```

Figure 6.13 A FORTRAN Implementation of a Recursive Loop Structure

The translator recognises the recursive nature of the loop and declares one member of the loop forward (in this case SD). It then declares procedures SB and SC in the main body of the program. Procedure SD is declared at this level even though no direct link exists between the main program and subroutine SD. SD is called by the two procedures SB and SC and its declaration at this higher level prevents it being duplicated. Function SE is only called by SD and

so it is declared within the bounds of SD. Its link with SB is not affected as SB has been declared previously. Figure 6.14 shows the translator output for the FORTRAN program of Figure 6.13.

Some versions of FORTRAN allow a subprogram to call itself recursively but this feature is non-standard. The translator handles this situation correctly and produces a Pascal subprogram which calls itself recursively.

```
PROGRAM FORTPASC;
        (*********)

VAR

   CONTRL : RECORD
      FLOW : BOOLEAN
   END;


PROCEDURE SD ;

FORWARD;

PROCEDURE SB ;

BEGIN
   IF ( CONTRL . FLOW ) THEN SD ;
END;

PROCEDURE SC ;

BEGIN
   SD ;
END;

PROCEDURE SD ;

VAR

   A , B : REAL ;

FUNCTION SE (B : REAL ) : REAL ;

BEGIN
   IF ( NOT CONTRL . FLOW ) THEN SB ;
   SE := B ;
END;


BEGIN
   A := SE ( B ) ;
END;

BEGIN
   CONTRL . FLOW := TRUE ;
   SB ;
   SC ;
END.
```

Figure 6.14 The Pascal Program Produced from Figure 6.13

6.2 Real Test Cases

At this point in time, the translator has been used on over 180 FORTRAN programs or subprograms and each one has been handled successfully by the translator. Either a correct Pascal program or procedure, or an error message highlighting an untranslatable FORTRAN feature, has been produced.

Apart from a small number of cases which were specifically designed to test a certain feature, all of the FORTRAN programs used have been taken from program libraries, written by other people or taken from the literature. This approach eliminates any personal bias of the author during the testing procedure. Four sources of FORTRAN subprograms proved particularly useful:

1) the problem section after Yourdon's chapter on Structured Programming [Yourdon 1975 pp 183-194]
2) the SYD11 assembler (a PDP 11/40 assembler written in FORTRAN for a large machine)
3) the Tektronix Terminal Control System subprogram library [Tektronix 1976]
4) a series of 10 programs and subprograms supplied by Dr B. A. Wichmann to test machine characteristics and library functions.

The SYD11 assembler and the Tektronix package have both been written in standard FORTRAN and are both easily portable systems. They were chosen because of their standard nature and to eliminate any problems of FORTRAN extensions which abound in many other FORTRAN program and subroutine libraries.

The examples taken from Yourdon were specifically designed to

test the structure handling procedure in the translator. Where a flow chart was given by Yourdon, a FORTRAN program was produced by the author and then it was translated to Pascal. In each case, a successful translation was carried out and a correct Pascal program was produced.

The SYD11 assembler contains a series of FORTRAN subroutines which handle the assembly and code producing sections of the assembler/simulator package. The driver program and the simulator itself are written in Algol and therefore only the assembler subroutines were converted by the translator. A complete Pascal version of SYD11 cannot be produced because the original program is not wholly written in FORTRAN. However, each subroutine represents a useful test for the translator. In total, the FORTRAN section represents about 4000 lines of code. Initially a dummy main program and a dummy simulator were added to the subroutine suite so that the subroutine ordering aspect of the translator could be tested. No problems were encountered in this area; the translator correctly identified the order and nesting of the FORTRAN subroutines.

As the translator proceeded to deal with each subroutine, a timing problem was encountered (see Chapter 7) and it proved expedient to split the FORTRAN program into smaller sections. A group of small subroutines was translated together and each of the large subroutines was translated individually. This approach avoided the timing problem encountered earlier.

The Tektronix Terminal Control System library contains 115 FORTRAN subprograms. A cursory glance through the library indicated

that 90% of the subroutines use a large COMMON block. About 60% of the subroutines use a simple straight line sequence of code, about 20% use a program flow structure which would not require the addition of extra variables and the remaining 20% use a complex flow structure requiring the addition of extra variables to convert them to a well structured form. These results indicate that the library contains more subroutines of a simple nature than the NAG Numerical Algorithms Library [Prudom & Hennell 1977]. These indications tend to confirm Prudom and Hennell's claim that many subprograms do not benefit from being manually recoded, and an automatic translation saves much human effort.

The subprograms are all written in standard FORTRAN and contain no I/O statements; those statements being contained in user written subroutines for each installation.

It proved expedient to split the library into small groups containing about 10 subprograms each and then submitting each group of subprograms to the translator. Each group of subprograms produced a set of correct Pascal procedures or functions. As the FORTRAN subprograms contained no I/O or non-standard features, the translator was able to produce correct Pascal procedures for each set of FORTRAN subprograms.

The series of programs supplied by Dr B. A. Wichmann were translated individually, altered where necessary, and then executed. In each case, it took only a few steps to transfer a program produced by the translator into a correct, executable program. The following discussion indicates the steps taken for one of this set of

programs.

The FORTRAN program, which is designed to test the SQRT function of a machine, was submitted to the translator and the translator produced the FORTRAN listing in Figure 6.15 and the Pascal listing in Figure 6.16. The FORTRAN listing contains a number of untranslatable conditions. The translator does not recognise the non-standard PROGRAM statement and it is ignored together with the associated error message, as the translator generates a program statement for the Pascal program.

This program uses a variable file identifier in FORTRAN as well as a constant file identifier. The Pascal output has a file declaration for the constant file but the buffer is only 68 characters long. The variable file identifier is set to write to the same file as the constant identifier in the FORTRAN program and that means that all references to the variable file identifier in I/O statements in the Pascal program must be changed to refer to the file identified by the constant identifier. The statement setting the variable identifier was deleted and the declaration of IOUT was removed from the Pascal declarations. A check was made to see whether the file buffer was now large enough. The new maximum size was found to be 71 characters and line 600 was altered accordingly.

The subprograms called by the program were translated separately and included in the compilation between lines 7400 and 7500. The program then compiled successfully and executed correctly.

This procedure was repeated for each of the other programs in the suite. In general, only a few steps, if any, are necessary to get a translated program compiled and executed.

```
$SET 3 DISK FORSTMT
      PROGRAM SORTEST(OUTPUT,TAPE6=OUTPUT)
 SCANNING :
>>>>>>0001>>>>>>UNRECOGNISED FORTRAN STATEMENT                                  E2022
C     DATA REQUIRED
C
C        NONE
C
C     OTHER SUBPROGRAMS IN THIS PACKAGE
C
C        MACHAR - AN ENVIRONMENTAL INQUIRY PROGRAM PROVIDING
C                 INFORMATION ON THE FLOATING-POINT ARITHMETIC
C                 SYSTEM.  NOTE THAT THE CALL TO MACHAR CAN
C                 BE DELETED PROVIDED THE FOLLOWING FIVE
C                 PARAMETERS ARE ASSIGNED THE VALUES INDICATED
C
C                 IBETA - THE RADIX OF THE FLOATING-POINT SYSTEM
C                 IT    - THE NUMBER OF THE BASE-IBETA DIGITS IN THE
C                         SIGNIFICAND OF A FLOATING-POINT NUMBER
C                 EPS   - THE SMALLEST POSITIVE FLOATING-POINT
C                         NUMBER SUCH THAT 1.0+EPS .NE. 1.0
C                 XMIN  - THE SMALLEST POSITIVE FLOATING-POINT
C                         NUMBER
C                 XMAX  - THE LARGEST FINITE FLOATING-POINT NO.
C
C        RANDL(X) - A FUNCTION SUBPROGRAM RETURNING LOGARITHMICALLY
C                   DISTRIBUTED RANDOM REAL NUMBERS.  IN PARTICULAR,
C                         A * RANDL(ALOG(B/A))
C                   IS LOGARITHMICALLY DISTRIBUTED OVER (A,B)
C
C           RAN(K) - A FUNCTION SUBPROGRAM RETURNING RANDOM REAL
C                    NUMBERS UNIFORMLY DISTRIBUTED OVER (0,1)
C
C     STANDARD FORTRAN SUBPROGRAMS REQUIRED
C
C        ABS, ALOG, EXP, FLOAT, INT, SQRT
C-----------------------------------------------------------------------
      INTEGER I,IBETA,IEXP,IOUT,IRND,IT,J,K,K1,MACHEP,MAXEXP,
     1        MINEXP,N,NEGEP,NGRD
      REAL A,ALBETA,B,BETA,C,EPS,EPSNEG,RANDL,R5,R6,R7,SQBETA,
     1     W,X,XMAX,XMIN,X1,X1,Y,Z
C
      IOUT = 6
C
      CALL MACHAR(IBETA,IT,IRND,NGRD,MACHEP,NEGEP,IEXP,MINEXP,
     1            MAXEXP,EPS,EPSNEG,XMIN,XMAX)
      BETA = FLOAT(IBETA)
      SQBETA = SQRT(BETA)
      ALBETA = ALOG(BETA)
      A = 1.0E0 / SQBETA
      B = 1.0E0
      N = 2000
C-----------------------------------------------------------------------
C     RANDOM ARGUMENT ACCURACY TESTS
C-----------------------------------------------------------------------
      DO 300 J = 1, 2
         C = ALOG(B/A)
         K = 0
         K1 = 0
         X1 = 0.0E0
         R5 = 0.0E0
         R6 = 0.0E0
         R7 = 0.0E0
C
         DO 200 I = 1, N
            X = A * RANDL(C)
            Y = X * X
            Z = SQRT(Y)
            W = (Z - Y) / X
            IF (W .GT. 0.0E0) K = K + 1
            IF (W .LT. 0.0E0) K1 = K1 + 1
```

```
                        R5 = W
                        X1 = X
                  120   R7 = R7 + W * W
                  200 CONTINUE
                C
                      XN = FLOAT(N)
                      R5 = R5 / XN
                      R7 = SQRT(R7/XN)
                      WRITE (IOUT,1000)
SCANNING : IOUT
>>>>>>0002>>>>>>VARIABLE FILE IDENTIFIERS NOT TRANSLATED TO PASCAL                                    E2040
                      WRITE (IOUT,1001) N,A,B
                      WRITE (IOUT,1002) K,K1
                      WRITE (6,1015) IT,IBETA
                      W = -999.0
                      IF (R5 .NE. 0.0) W = ALOG(ABS(R5))/ALBETA
                      WRITE (IOUT,1003) R5,IBETA,W
                      X = -999.0
                      IF (R6 .NE. 0.0) X = ALOG(ABS(R6))/ALBETA
                      WRITE (IOUT,1004) R6,IBETA,W,X1
                      W = -999.0
                      IF (R7 .NE. 0.0) W = ALOG(ABS(R7))/ALBETA
                      WRITE (IOUT,1005) R7,IBETA,W
                      A = 1.0E0
                      B = SQBETA
                  300 CONTINUE
                C-----------------------------------------------------------------------
                C     SPECIAL TESTS
                C-----------------------------------------------------------------------
                      WRITE (IOUT,1009)
                      X = XMIN
                      Y = SQRT(X)
                      WRITE (IOUT,1010) X,Y
                      X = 1.0E0 - EPSNEG
                      Y = SQRT(X)
                      WRITE (IOUT,1011) EPSNEG,Y
                      X = 1.0E0
                      Y = SQRT(X)
                      WRITE (IOUT,1012) X,Y
                      X = 1.0E0 + EPS
                      Y = SQRT(X)
                      WRITE (IOUT,1013) EPS,Y
                      X = XMAX
                      Y = SQRT(X)
                      WRITE (IOUT,1014) X,Y
                C-----------------------------------------------------------------------
                C     TEST OF ERROR RETURNS
                C-----------------------------------------------------------------------
                      WRITE (IOUT,1006)
                      X = 0.0E0
                      WRITE (IOUT,1007) X
                      Y = SQRT(X)
                      WRITE (IOUT,1008) Y
                      X = -1.0E0
                      WRITE (IOUT,1007) X
                      Y = SQRT(X)
                      WRITE (IOUT,1008) Y
                      WRITE (IOUT,1021)
                      STOP
                 1000 FORMAT(22H1TEST OF SQRT(X*X) - X //)
                 1001 FORMAT(17,47H RANDOM ARGUMENTS WERE TESTED FROM THE INTERVAL /
                     1 9X,1H(,E15.4,1H,,E15.4,1H)//)
                 1002 FORMAT(30H THE SQUARE ROOT WAS TOO LARGE,I6,11H TIMES, AND /
                     1 9X,9HTOO SMALL,I6,7H TIMES.//)
                 1003 FORMAT(25H THE MEAN RELATIVE ERROR WAS,E15.4,3H = ,I4,3H **,
                     1 F7.2//)
                 1004 FORMAT(30H THE MAXIMUM RELATIVE ERROR OF,E15.4,3H = ,I4,3H **,
                     1 F7.2/4X,10HOCCURRED,10H FOR X = ,E17.6//)
                 1005 FORMAT(40H THE ROOT MEAN SQUARE RELATIVE ERROR WAS,E15.4,
                     1 3H = ,I4,3H **,F7.2)
                 1006 FORMAT(22H1TEST OF ERROR RETURNS //)
                 1007 FORMAT(36H SQRT WILL BE CALLED WITH THE ARGUMENT,E15.4//)
                 1008 FORMAT(24H SQRT RETURNED THE VALUE,E15.4///)
                 1009 FORMAT(26H1TEST OF SPECIAL ARGUMENTS//)
                 1010 FORMAT(19H SQRT(XMIN) = SQRT(,E15.7,4H) = ,E15.7//)
                 1011 FORMAT(25H SQRT(1-EPSNEG) = SQRT(1-,E15.7,4H) = ,E15.7//)
                 1012 FORMAT(18H SQRT(1.0) = SQRT(,E15.7,4H) = ,E15.7//)
```

```
 1015 FORMAT(10H THERE ARE,I4,5H BASE,I4,
     1    45H SIGNIFICANT DIGITS IN A FLOATING-POINT NUMBER  ///)
 1021 FORMAT(25H THIS CONCLUDES THE TESTS )
C     ---------- LAST CARD OF SQRT TEST PROGRAM ----------
      END
      MACHAR
 SCANNING :
>>>>>>0003>>>>>>ABOVE SUBROUTINE/FUNCTION NOT INCLUDED IN TRANSLATION                    E2150
      SANDL
 SCANNING :
>>>>>>0004>>>>>>ABOVE SUBROUTINE/FUNCTION NOT INCLUDED IN TRANSLATION                    E2150

 ---------- END OF FORTRAN LISTING ----------
156 FORTRAN CARDS READ, 167 FORTRAN LINES WRITTEN
4 ERRORS FOUND BY THE TRANSLATOR
```

Figure 6.15 FORTRAN Listing of the SQRT Testing Program

```
PROGRAM FORTPASC;                                                       00000100
      (********)                                                        00000200
                                                                        00000300
VAR                                                                     00000400
                                                                        00000500
   FORFIL66 : FILE OF PACKED ARRAY [1 .. 68] OF CHAR;                   00000600
                                                                        00000700
(*     DATA REQUIRED                                                    00000800
                                                                        00000900
          NONE                                                          00001000
                                                                        00001100
     OTHER SUBPROGRAMS IN THIS PACKAGE                                  00001200
                                                                        00001300
        MACHAR - AN ENVIRONMENTAL INQUIRY PROGRAM PROVIDING             00001400
                 INFORMATION ON THE FLOATING-POINT ARITHMETIC           00001500
                 SYSTEM.  NOTE THAT THE CALL TO MACHAR CAN              00001600
                 BE DELETED PROVIDED THE FOLLOWING FIVE                 00001700
                 PARAMETERS ARE ASSIGNED THE VALUES INDICATED           00001800
                                                                        00001900
                 IBETA  - THE RADIX OF THE FLOATING-POINT SYSTEM        00002000
                 IT     - THE NUMBER OF THE BASE-IBETA DIGITS IN THE    00002100
                          SIGNIFICAND OF A FLOATING-POINT NUMBER        00002200
                 EPS    - THE SMALLEST POSITIVE FLOATING-POINT          00002300
                          NUMBER SUCH THAT 1.0+EPS .NE. 1.0             00002400
                 XMIN   - THE SMALLEST POSITIVE FLOATING-POINT          00002500
                          NUMBER                                        00002600
                 XMAX   - THE LARGEST FINITE FLOATING-POINT NO.         00002700
                                                                        00002800
        RANDL(X) - A FUNCTION SUBPROGRAM RETURNING LOGARITHMICALLY      00002900
                   DISTRIBUTED RANDOM REAL NUMBERS.  IN PARTICULAR,     00003000
                          A * RANDL(ALOG(B/A))                          00003100
                   IS LOGARITHMICALLY DISTRIBUTED OVER (A,B)            00003200
                                                                        00003300
        RAN(K) - A FUNCTION SUBPROGRAM RETURNING RANDOM REAL            00003400
                 NUMBERS UNIFORMLY DISTRIBUTED OVER (0,1)               00003500
                                                                        00003600
                                                                        00003700
     STANDARD FORTRAN SUBPROGRAMS REQUIRED                              00003800
                                                                        00003900
       ABS, ALOG, EXP, FLOAT, INT, SQRT                                 00004000
  ------------------------------------------------------------------*)  00004100
                                                                        00004200
                                                                        00004300
  I , IBETA , IEXP , IOUT , IRND , IT , J , K , K1 , MACHEP , MAXEXP ,  00004400
  MINEXP , N , NEGEP , NGRD : INTEGER ;                                 00004500
  A , ALGBETA , B , BETA , C , EPS , EPSNEG , R5 , R6 , R7 , SQBETA , W 00004600
  , X , XMAX , XMIN , XN , X1 , Y , Z : REAL ;                          00004700
                                                                        00004800
                                                                        00004900
  FORMAT                                                                00005000
                                                                        00005100
    FORM1000 ("TEST OF SQRT(X*X) - X"//);                               00005200
    FORM1001 (I7," RANDOM ARGUMENTS WERE TESTED FROM THE INTERVAL"/X5,  00005300
    "(",E15.4,",",E15.4,")"//);                                         00005400
    FORM1002 ("THE SQUARE ROOT WAS TOO LARGE",I6," TIMES, AND"/X3,      00005500
    "TOO SMALL",I5," TIMES."//);                                        00005600
    FORM1015 ("THERE ARE",I4," BASE",I4,                                00005700
    " SIGNIFICANT DIGITS IN A FLOATING-POINT NUMBER"//);                00005800
    FORM1003 ("THE MEAN RELATIVE ERROR WAS",E15.4," = ",I4," **",F7.2//); 00005900
    FORM1004 ("THE MAXIMUM RELATIVE ERROR OF",E15.4," = ",I4," **",F7.2/X 00006000
    3,"OCCURRED FOR X =",E17.8//);                                      00006100
    FORM1005 ("THE ROOT MEAN SQUARE RELATIVE ERROR WAS",E15.4," = ",I4, 00006200
    " **",F7.2);                                                        00006300
    FORM1009 ("TEST OF SPECIAL ARGUMENTS"//);                           00006400
    FORM1010 ("SQRT(XMIN) = SQRT(",E15.7,") = ",E15.7//);               00006500
    FORM1011 ("SQRT(1-EPSNEG) = SQRT(1-",E15.7,") = ",E15.7//);         00006600
    FORM1012 ("SQRT(1.0) = SQRT(",E15.7,") = ",E15.7//);                00006700
    FORM1013 ("SQRT(1+EPS) = SQRT(1+",E15.7,") = ",E15.7//);            00006800
    FORM1014 ("SQRT(XMAX) = SQRT(",E15.7,") = ",E15.7//);               00006900
    FORM1006 ("TEST OF ERROR RETURNS"//);                               00007000
    FORM1007 ("SQRT WILL BE CALLED WITH THE ARGUMENT",E15.4//);         00007100
    FORM1008 ("SQRT RETURNED THE VALUE",E15.4//);                       00007200
    FORM1021 ("THIS CONCLUDES THE TESTS");                              00007300
                                                                        00007400
  BEGIN                                                                 00007500
    IOUT := 6 ;                                                         00007600
                                                                        00007700
```

```
                IBETA := ( IBETA );
                SQBETA := SQRT ( IBETA );
                ALBETA := LN ( IBETA );
                A := 1.0E0 / SQBETA ;
                X := 1.0E0 ;
                Y := 2.0E0 ;
        (*--------------------------------------------------------------------
                RANDOM ARGUMENT ACCURACY TESTS
        --------------------------------------------------------------------*)
                FOR J := 1 TO 2 DO BEGIN
                    C := LN ( J / A );
                    K := 0 ;
                    K1 := 0 ;
                    X1 := 0.0E0 ;
                    R5 := 0.0E0 ;
                    R6 := 0.0E0 ;
                    R7 := 0.0E0 ;

                    FOR I := 1 TO N DO BEGIN
                        A := A * RANDL ( C ) ;
                        Y := X * X ;
                        Z := SQRT ( Y );
                        W := ( Z - X ) / X ;
                        IF ( W > 0.0E0 ) THEN K := K + 1 ;
                        IF ( W < 0.0E0 ) THEN K1 := K1 + 1 ;
                        R5 := R5 + W ;
                        W := ABS ( W );
                        IF NOT ( W <= R6 ) THEN BEGIN
                            R6 := W ;
                            X1 := X ;
                        END;
                        R7 := R7 + W * W ;
                    END ;

                    XN := ( N );
                    R5 := R5 / XN ;
                    R7 := SQRT ( R7 / XN );
                    WRITE (IOUT , FORM1000 );
                    WRITE (IOUT , FORM1001 , BEGIN K ; A ; B ; END );
                    WRITE (IOUT , FORM1002 , BEGIN K ; K1 ; END );
                    WRITE (OUTFILE6 , FORM1015 , BEGIN IT ; IBETA ; END );
                    W := - 999.0 ;
                    IF ( R5 <> 0.0 ) THEN W := LN ( ABS ( R5 ))/ ALBETA ;
                    WRITE (IOUT , FORM1003 , BEGIN R5 ; IBETA ; W ; END );
                    W := - 999.0 ;
                    IF ( R6 <> 0.0 ) THEN W := LN ( ABS ( R6 ))/ ALBETA ;
                    WRITE (IOUT , FORM1004 , BEGIN R6 ; IBETA ; W ; X1 ; END );
                    W := - 999.0 ;
                    IF ( R7 <> 0.0 ) THEN W := LN ( ABS ( R7 ))/ ALBETA ;
                    WRITE (IOUT , FORM1005 , BEGIN R7 ; IBETA ; W ; END );
                    A := 1.0E0 ;
                    A := SQBETA ;
                END ;
        (*--------------------------------------------------------------------
                SPECIAL TESTS
        --------------------------------------------------------------------*)
                WRITE (IOUT , FORM1009 );
                X := XMIN ;
                Y := SQRT ( X );
                WRITE (IOUT , FORM1010 , BEGIN X ; Y ; END );
                X := 1.0E0 - EPSNEG ;
                Y := SQRT ( X );
                WRITE (IOUT , FORM1011 , BEGIN EPSNEG ; Y ; END );
                X := 1.0E0 ;
                Y := SQRT ( X );
                WRITE (IOUT , FORM1012 , BEGIN X ; Y ; END );
                X := 1.0E0 + EPS ;
                Y := SQRT ( X );
                WRITE (IOUT , FORM1013 , BEGIN EPS ; Y ; END );
                X := XMAX ;
                Y := SQRT ( X );
                WRITE (IOUT , FORM1014 , BEGIN X ; Y ; END );
        (*--------------------------------------------------------------------
                TEST OF ERROR RETURNS
        --------------------------------------------------------------------*)
                WRITE (IOUT , FORM1006 );
                X := 0.0E0 ;
                WRITE (IOUT , FORM1007 , BEGIN X ; END );
```

```
X := - 1.000 ;
WRITE (IOUT , FORM1007 , BEGIN X ; END );
Y := SQRT ( X );
WRITE (IOUT , FORM1008 , BEGIN Y ; END );
WRITE (IOUT , FORM1021 );
(*  ----------LAST CARD OF SQRT TEST PROGRAM ----------  *)
END.
```

```
00015900
00016000
00016100
00016200
00016300
00016400
00016500
00016600
```

```
========== END OF PASCAL LISTING ==========
160 PASCAL LINES WRITTEN
PASCAL DISK FILE CREATED - TITLE = (USISC009)PDISK ON PACK.
```

Figure 6.16 The Pascal Program Produced by the Translator for the SQRT Testing Program

7. EVALUATION OF RESULTS

In Chapter 6, a number of examples were looked at in detail and some 'real' programs were examined. To date, the translator has been used on over 180 FORTRAN programs and subprograms and in each case a workable Pascal program or procedure has been produced. A number of these Pascal programs have been successfully compiled and executed after translation.

The translator itself was written in B6700 Algol [B6700 Algol 1977] because at the time the project was started the B6700 Pascal compiler had not been fully developed and the limitations then imposed on its use were too severe for it to be considered for this project. The translator also makes use of some of the B6700 Algol fault handling facilities to interrogate the status of program variables and arrays when a fault occurs. These facilities are included according to the setting of a compile time option in the translator. No such facilities exist in B6700 Pascal.

The translator is comparable to a compiler in terms of its size. It contains over 14000 lines of code and during execution it occupies an average of 11180 words of core for code and an average of 5380 words for data. The number of words required for code depends on the depth of recursion necessary to generate a correct program, and the number required for data depends on the size of the translator's heap, which depends on the number of variables and blocks in the FORTRAN subprogram.

The time taken during execution varies according to the number

of blocks and the number of loops in each FORTRAN subprogram. Simple cases, such as those discussed in Section 6.1, take 3 to 5 seconds CPU time on the B6700. Larger subprograms take longer and, as an example, a group of subroutines amounting to 400 lines of FORTRAN code were translated in 30 seconds. During the translation of simple or straightforward cases, the translator spends about 50% of its time in the procedure which reads the FORTRAN program and builds the internal tables. In these cases, about 2% of its time is spent in the procedure which finds all the loops of a graph.

The translator uses an algorithm, due to Tiernan [Tiernan 1970], to find all the circuits in a graph. As Tiernan suggests in his paper, the algorithm is "theoretically most efficient" but it is costly to run on a graph with a high arc-to-vertex density and containing more than 50 vertices. Tiernan gives the expression

$$T < n \left[1 + \int_{1}^{M} \top'(x+1)\, dx \right]$$

where T is the time spent,

n is a factor relating to computer computation speed,

M is the number of arcs,

to show the relationship between the time taken to discover all the loops and the number of arcs in a graph.

When cases with a high arc-to-vertex density were submitted to the translator, the time spent in the loop detection procedure increased significantly to the point where it became the dominant procedure in terms of time. These findings tend to verify Tiernan's suggestion of a practical limit of 100 arcs in a graph.

The Pascal listings produced by the translator are generally longer than the corresponding FORTRAN listings. This is primarily due to the nature of the two languages in that FORTRAN is a compact language whereas Pascal encourages a programmer to set his program out in a style he finds suitable. The stylistics used by the translator tend to encourage an increase in program length. Blank lines are used to separate the type, var, etc. sections of the program. Indentation is used and a space is inserted after each identifier so many expressions overflow the available character positions on a line and force new lines to be generated. Tests show that, on average, the Pascal programs produced by the translator contained 1.8 times the number of lines of the FORTRAN programs.

Only in one case was the number of lines produced less than the FORTRAN source; that case involved a suite of subroutines, each with a large COMMON block and little executable code. The FORTRAN source contained the COMMON block in each subprogram but in the Pascal program produced the COMMON block was moved to the outer block and only declared once.

The input/output conversion has many drawbacks. A number of problems exist but the main ones are that FORTRAN I/O is record-oriented and the number of variables in an I/O list is determined at run-time. In Pascal, I/O is stream-oriented and each I/O statement must contain a fixed number of variables. As stated in section 3.2.1, the problem is generally unsolvable and the approach adopted by the translator is heavily biased to a Pascal extension available in B6700 Pascal [B6700 Pascal 1978]. The approach is

justified because the main use envisaged for the translator will be to translate FORTRAN subroutine libraries, very few of which employ any I/O; and because it was felt that the use of an extension available in a version of Pascal was better than making no attempt to translate I/O. The availability of fast editing facilities on most modern computers should make the task of altering any Pascal program produced an easy one. The example in section 6.2 gives an indication of the process involved in changing a translated program into an executable Pascal program.

The removal of all COMMON blocks to the outer level and the record structures used in Pascal for COMMON and EQUIVALENCE statements produce a clear picture of these FORTRAN structures. The correspondence between variables used in EQUIVALENCE statements, and variables with different types or names in COMMON statements, is apparent.

The efficiency of the Pascal programs produced may be slightly less than the corresponding FORTRAN programs during execution, if efficiency is measured in terms of the number of instructions executed. The translator uses extra variables to control the flow of some programs and extra code is added to set and test these variables. For medium or large programs, the addition of a small number of variables does not add significantly to the program's size, nor to the execution time. For very small programs, the addition of a variable may be significant.

It is difficult to draw conclusions when comparing execution details of the two forms of a program because the details depend on

the machine being used, on the efficiency of the code produced by the relevant compilers and on the size of the program in question. However, on the Burroughs B6700, the FORTRAN and Pascal compilations take approximately the same time and there is no significant difference in execution time for the two forms.

While there may be a slight loss of efficiency, the benefits gained should outweigh the losses. These benefits include the production of better code via flow analysis, and the production of a clearer, structured program which should be easier to understand and maintain. Efficiency has become a somewhat academic point in these days of large generalised operating systems and high-level languages, and it seems that an order of magnitude in programmer productivity is worth a little overhead.

8. CONCLUSION

The work described in the previous sections and the results discussed indicate that the concept of translating FORTRAN to Pascal is a viable one. The limitations imposed by the languages and by the translator appear to affect a minority of cases and these limitations are far outweighed by the advantages to be gained by performing the translation automatically and producing programs in a structured language. The translation process can be regarded as a move from a low-level language to a language of a higher level.

The translator should gain most of its use from translating the vast amount of subprogram material available in FORTRAN subroutine libraries to Pascal. To that end, it will be a valuable aid in moving the scientific computer industry away from its dependence on FORTRAN and encouraging the use of the structured language, Pascal. However, should the Pascal language fail in its attempt to become an industry standard, the principles of the translation process and the structure of the translator indicate that it would be a relatively straightforward process to modify the translator so that it produced programs in another structured language.

Many conversion problems still exist with the FORTRAN - Pascal translator. These problems include the difficulties with the input/output conversion, the inability of the Pascal language to accept the types COMPLEX and DOUBLE PRECISION and the lack of a facility in Pascal to link in externally compiled subprograms. The last two difficulties mentioned will probably hamper the rise in popularity of Pascal but they are not insurmountable and will

probably be included in the language in the future.

APPENDIX

A. THE EFFECT OF THE NEW FORTRAN STANDARD ON THE TRANSLATOR

The draft proposed ANS FORTRAN standard was published in the SIGPLAN Notices in 1976 [Fortran 1976]. Since then the standards committee has accepted a number of changes to the language and in April 1978 the standard was approved [ANSI 1978]. This appendix looks at the effects that the new standard will have on the conversion of FORTRAN 77 (as the new standard is commonly called) to Pascal. A number of authors, including [Woolley 1977] and [Brainerd 1978], have examined the effects of the changes on the language but this section will be concerned with the conversion aspect only.

Figure A.1 shows the FORTRAN 77 statements with no significant changes from FORTRAN 66 [Fortran 1973]. The type declaration statements, shown in the figure, do have significant changes when used as array declarators and these are examined later. The ASSIGN statement is now permitted to be used with FORMAT statements and it is possible to translate this feature in a manner similar to the way the ASSIGN and assigned GOTO statements are currently handled (see 4.5.1.2 and 4.5.1.4). For example, the FORTRAN program segment:

```
      10 FORMAT ( ... )
      20 FORMAT ( ... )
           .
         ASSIGN 10 TO I
           .
         ASSIGN 20 TO I
           .
         WRITE (6,FMT=I) ...
```

would become in Pascal

```
         format
           FORMAT10( ... );
           FORMAT20( ... );
             .
           I := 1;
             .
           I := 2;
             .
           case I of
           1:
              write(FORFIL06,FORMAT10, ... );
           2:
              write(FORFIL06,FORMAT20, ... );
           end;
```

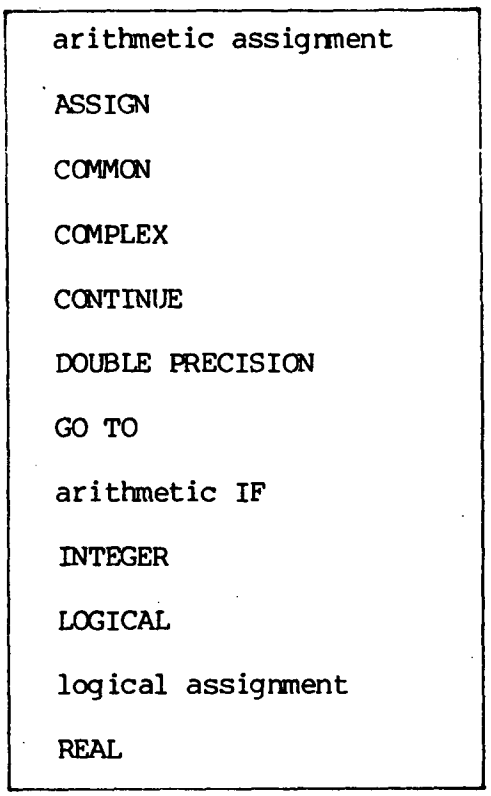


Figure A.1 FORTRAN 77 Statements with no Significant Changes

Figure A.2 shows the FORTRAN 77 statements with minor changes. The SUBROUTINE statement now permits alternative RETURNs to be specified and this feature will be examined later. The FORTRAN END statement is now permitted to act as a RETURN or a STOP statement - a feature which should not cause any trouble to the translator as it stands. The changes to the other statements shown in Figure A.2 should not significantly affect the translation process.

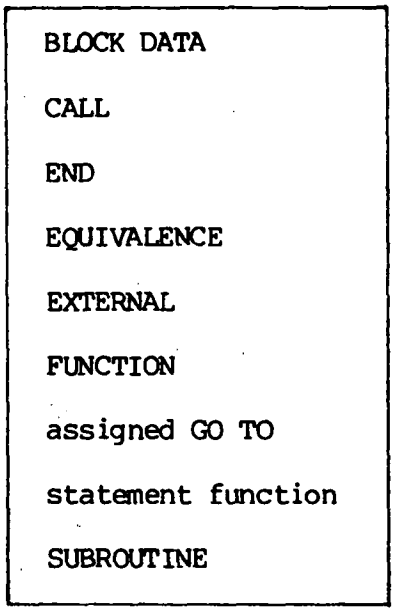


Figure A.2 FORTRAN 77 Statements with Minor Changes

Figure A.3 contains a list of FORTRAN statements to which significant changes have been made. Of these statements, the BACKSPACE, DATA, ENDFILE, FORMAT, computed GO TO, PAUSE, REWIND and STOP statements would be handled in a manner similar to that of FORTRAN 66.

The DIMENSION statement is now permitted to declare arrays of up to 7 dimensions and each array may specify its lower bound explicitly. This facility is permitted in Pascal, and Pascal allows an array to consist of an unrestricted number of dimensions. The removal of some of the FORTRAN 66 restrictions on array declarations results in a simplification of the translation process. FORTRAN, however, has been extended to allow expressions as dimension bounds but this facility is not permitted in Pascal.

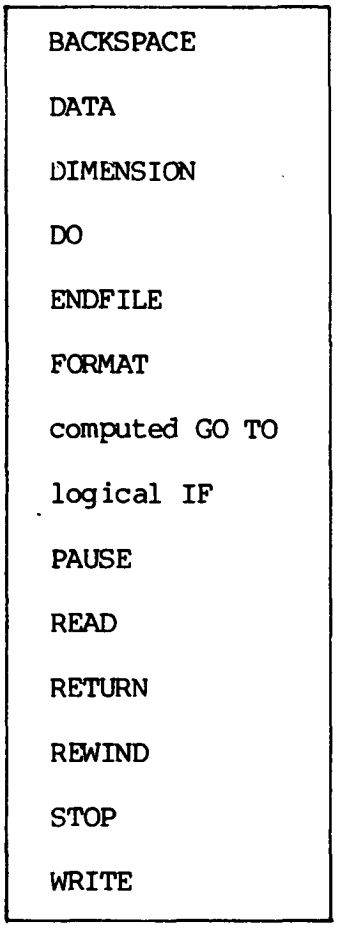


Figure A.3 FORTRAN Statements with Major Changes

The DO statement has been altered in a number of ways. Firstly, expressions are permitted for the initial, final and increment values but they should not significantly affect the translation. The final value is now permitted to be less than the initial value and where this occurs, the loop is not entered. The new standard defines an iteration count which is calculated from the three DO expressions before the loop is entered. To cater for this situation, the translator must generate a _while_ loop in Pascal rather than the _for_ or _repeat ... until_ constructs currently generated. The _for_ loop in Pascal leaves the control variable undefined on exiting the loop and the _repeat ... until_ loop executes the loop at least once. In

general, then, the FORTRAN statement

DO <label> I = $expr_1$,$expr_2$,$expr_3$

becomes in Pascal

```
I := expr1 ;
LIMITV01 := expr2 ;
INCVAR01 := expr3 ;
TRIPVA01 := max(trunc((LIMITV01-I+INCVAR01)/INCVAR01),0);
while ( TRIPVA01 > 0) do begin
    .
    .
    .
   I := I + INCVAR01;
   TRIPVA01 := TRIPVA01-1;
end;
```

where

(1) LIMITV01 ("LIMIT Variable") and INCVAR01 ("INCrementation VARiable") are new variables of the same type as I and their names are generated in accordance with section 4.1.6. They are necessary because in FORTRAN 77 all of the DO expressions are evaluated before the loop is entered.

(2) TRIPVA01 ("TRIP VAriable") is an integer variable used to control the number of times the loop is executed.

A number of neater translations could be given under certain conditions. If the incrementation expression did not change during the range of the loop, and it did not contain a function with a side effect, the new variable, INCVAR01, could be eliminated and the statement

```
I := I + INCVAR01;
```

replaced by

```
I := I + expr3;
```

Similarly, if the limit expression ($expr_2$) was not altered during the range of the DO loop, and it did not contain a function with a side effect, the new limit variable and trip counter could be eliminated and the whole loop replaced by the Pascal statements

```
I := expr1;
while (I <= expr2) do begin
   .
   .
   .
  I := I + expr3;
end;
```

These statements would probably be the most commonly used translation but the previous general translation must be permitted to cater for all possibilities. It would be possible to retain the use of the FORSTMT translator option (see 4.8.6) and generate a Pascal *for* statement from a FORTRAN DO statement with an incremental value of 1. The constraint of not using the control variable after the loop would have to be emphasized, but the requirement that the initial value be not less than the final value would be relaxed.

The logical IF statement has been altered to permit an IF ... THEN ... ELSE ... construct. This change should convert to the Pascal *if* statement in a straightforward manner.

A number of significant changes have been made to the FORTRAN READ and WRITE statements. The statements may omit the unit

identifier and a default file is assumed. This change can convert directly to Pascal using the files FORFIL05 and FORFIL06 as the defaults (see 4.5.1.14). The Pascal files INPUT and OUTPUT are not suitable for use as the default files because, in FORTRAN, the omission or the specification of the default unit number (5 or 6) refers to the same file.

eg. WRITE(FMT=99) ...

WRITE(6,FMT=99) ...

A number of specifiers are permitted in these statements to determine the unit number, FORMAT statement, record number and to handle exception conditions. Apart from handling the FORMATs in the manner indicated previously (see 4.5.2.8) and translating constant unit identifiers only, the Pascal language cannot handle FORTRAN specifiers. List-directed input and output is available in FORTRAN 77 and this facility probably represents the greatest advance in the translation of I/O from FORTRAN to Pascal. Apart from a few (minor) inconsistencies, FORTRAN list-directed I/O may be translated to Pascal stream-oriented I/O. The inconsistencies include the special handling of the slash character in FORTRAN 77 and the way Pascal handles end-of-line on text files. FORTRAN 77 allows core-to-core I/O using READ and WRITE and permits the use of direct-access files but neither facility is available in Pascal. A paper by Sale [Sale 1978c] suggests a way in which Pascal could be extended to handle core-to-core I/O. If this approach were taken by a compiler, it would be possible for the translator to generate this sort of Pascal core-to-core I/O operation.

The RETURN statement in FORTRAN 77 permits the use of alternate

returns, ie. the statement following the call to a subroutine is not necessarily the next statement to be executed in the calling subprogram after the return from the subroutine. In Pascal, each procedure has one exit only but it is possible to achieve the effect of alternate returns in FORTRAN using the following method:

(1) Ignore all alternate return specifiers in subroutine declarations but add an additional integer var parameter to the Pascal procedure declaration.

(2) When a RETURN statement is encountered in the subroutine set the additional integer parameter to the value of the integer expression in the RETURN statement, if the expression is present. If no expression exists, set the parameter to zero. Continue to translate the RETURN statement in the manner used for FORTRAN 66 (see 4.5.1.9).

(3) The additional variable must be declared in the calling subprogram. Translate a CALL statement in the manner used for FORTRAN 66 (see 4.5.1.8) but ignore all alternate return specifiers and include the new variable in the parameter list. After the procedure invocation statement add additional if statements (or a case statement) to transfer control to the required label depending on the value of the new variable.

For example, the FORTRAN 77 program:

```
SUBROUTINE S (I,*,J)
   .
RETURN
   .
RETURN 1
END
   .
   .
CALL S (IA,*123,IB)
   .
```

becomes in Pascal

```
var
   RETURN01 : integer;    {new variable}
   .
procedure S (I,J : integer; var RETURNVA : integer);
   .
   RETURNVA := 0;   goto 999;    {normal return}
   .
   RETURNVA := 1;   goto 999;    {alternate return}
999: end;
   .
   .
S(IA,IB,RETURN01);
if (RETURN01 = 1) then goto 123;
   .
```

If more than one alternate RETURN is specified a case statement after the procedure invocation would be more appropriate and, in

either case, the goto statement may be eliminated by the reordering of statements.

CHARACTER
character assignment
CLOSE
ENTRY
IMPLICIT
INQUIRE
INTRINSIC
OPEN
PARAMETER
PRINT
PROGRAM
SAVE

Figure A.4 New FORTRAN 77 Statements

Figure A.4 illustrates the statements which are new in FORTRAN 77. Of these statements, the CLOSE, INQUIRE and OPEN statements cannot be translated, in general, to Pascal. The CLOSE statement may be translated to the B6700 Pascal CLOSE statement but this is not standard Pascal.

The INQUIRE statement is used to determine certain properties of a file. No equivalent statement exists in Pascal.

Under a set of conditions, the FORTRAN OPEN statement may be ignored because Pascal performs an implicit OPEN when it attempts the first access on a file. These conditions would include the unit

number and title being constant.

One of the most significant changes in standard FORTRAN is the addition of a character data type and, in general, this data type would convert to the Pascal type, char. FORTRAN declarations of the form

```
CHARACTER * 80 LINE
```

could be converted to the Pascal form

```
LINE : packed array [1..80] of char;
```

However, this translation would create problems in assignment statements in a Pascal program. FORTRAN 77 will pad or truncate strings in an assignment statement to fit the size of the variable concerned. Standard Pascal requires the variable and expression to be of the same type, although many versions perform string padding. FORTRAN also permits a character array to have a lower bound not equal to one. This would create problems in Pascal in the assignment of a string to an array of this type. The concatenation operator, new in FORTRAN, is not available in Pascal. However, a procedure to perform a concatenation function could be provided, but a different procedure for each combination of character string lengths used by the concatenation operator in the FORTRAN program would have to be provided - a possible, but untidy translation.

Character strings may be returned as the value of a function in FORTRAN, but in Pascal the type of a function must be a scalar or a subrange. If the suggestions of [Sale 1978c] were adopted in Pascal, character functions could be converted directly to Pascal. In the mean time, functions returning a string of length 1 can be converted

directly to Pascal. FORTRAN functions returning a string of length greater than 1 could be replaced by a procedure in Pascal with an extra parameter to return the function value. Then each statement which used the character function would be translated to a procedure call of the Pascal procedure, followed by the translated version of the FORTRAN statement with the extra procedure parameter variable substituting for the function call.

Some additional intrinsic functions for character handling are available in FORTRAN. The LEN function cannot be converted to Pascal but the functions ICHAR and CHAR could be, although they would probably be inefficient. FORTRAN character substring references cannot, in general, be translated to Pascal.

The ENTRY statement has no equivalent in Pascal. However, it is possible to translate subprograms containing an ENTRY statement to Pascal. The process involves generating a subprogram for each ENTRY statement, making all the common data global and translating all the executable statements which are reachable from that entry point. An algorithm would have to be devised to determine the blocks of statements which could not be reached from that entry point and these blocks could be omitted from the corresponding subprogram.

The IMPLICIT statement, often used as an extension to FORTRAN 66, has been included in FORTRAN 77. It has been added to the translator (see 4.7.1) and the new standard definition is already catered for.

The new INTRINSIC statement in FORTRAN is used to allow the passing of an intrinsic function as an argument. It is required

since the use of an EXTERNAL statement on an intrinsic function destroys its definition as an intrinsic function. No such anomaly exists in Pascal and so it should be possible for the translator to ignore this statement.

The FORTRAN PARAMETER statement is used to give a symbolic name to a constant and it should translate to the const section of a Pascal subprogram without any problems.

The PRINT statement allows formatted output to the standard output device and it would be handled by the translator in a manner similar to that for the WRITE statement.

The PROGRAM statement is optional in FORTRAN 77 but, when it is used, the translator would be able to use the program name from the statement in the Pascal program statement instead of generating a name.

The SAVE statement is used to provide a means of saving the values of local variables, local arrays and named COMMON blocks between calls to a subroutine or function. No such facility existed in FORTRAN 66 although a large number of processors implemented it for all local variables. It is not possible to convert this statement directly to Pascal but by moving all the variables out of the procedure to a global area the effect may be achieved. To overcome any possible problems with duplicate variable names, the variables could be given a unique name or placed in a record and the record given a unique name - possibly generated from the subprogram name (see 4.1.6). The placing of these variables in a record offers

a more general solution as it eliminates the possibility of two identifiers with the same name in different subprograms with similar names forming the same identifier in Pascal. Any variable in a named COMMON block would be omitted from the record as it is already located in the Pascal global area (see 4.5.2.3), and any reference to a SAVEd item in the subprogram would be replaced by a reference to the corresponding item in the global record in Pascal.

eg. The FORTRAN statements

```
SUBROUTINE RANDOM (X)
INTEGER SEED
SAVE SEED
X = SEED
   .
   .
```

become in Pascal

```
program ...
   .
   .
var
   RANDOMSA : record      {a unique record name}
      SEED : integer
   end;
   .
   .
procedure RANDOM (var X : real);
begin
   X := RANDOMSA.SEED;
   .
```

In summary, this appendix has given a brief explanation of how the translator would handle FORTRAN programs written according to the new FORTRAN standard [ANSI 1978]. Much of the fine detail has been omitted to keep this appendix small, but the changes to statements have been examined in sufficient detail to indicate that the translation is equally possible. Some significant improvements to FORTRAN (eg. the type CHARACTER, list-directed I/O) simplify the translation process, but statements such as ENTRY add additional difficulties to the translator.

BIBLIOGRAPHY

(Abrahams 1975)

Abrahams, P., 'Structured Programming' Considered Harmful, SIGPLAN Notices, vol. 10, no. 4, April 1975, pp 13-24

(Aho & Ullman 1973)

Aho, A.V. & Ullman, J.D., The Theory of Parsing, Translation and Compiling, Prentice-Hall, 1973

(Allen 1971)

Allen, F.E., A Basis for Program Optimization, Proceedings of IFIP Congress 1971, Information Processing 71, North-Holland Publishing Company (1972), pp 385-390

(ANSI 1978)

American National Standards Institute, American National Standard Programming Language FORTRAN, ANSI X3.9-1978, American National Standards Institute, 1978

(Ashcroft & Manna 1971)

Ashcroft, E. & Manna, Z., The Translation of 'Goto' Programs to 'While' Programs, Computer Science Department, Stanford University, Report No. STAN-CS-71-188, January 1971

(B6700 Algol 1977)

Burroughs B6700 Algol Reference Manual, Burroughs Ltd., Report No. 5001639, May 1977

(B6700 Pascal 1978)

B6700/B7700 Pascal Reference Manual, Report No. R77-3, Department of Information Science, University of Tasmania, April 1978

(Bachmann 1971)

Bachmann, P., A Contribution to the Problem of the Optimization of Programs, Proceedings of IFIP Congress 1971, Information Processing 71, North-Holland Publishing Company (1972), pp 397-401

(Baker 1972)

Baker, F.T., Chief Programmer Team Management of Production Programming, I.B.M. Systems Journal, no. 1, 1972, pp 56-73

(Ballard & Tsichritzis 1971)

Ballard, A. & Tsichritzis, D., Transformations of Programs, Proceedings of IFIP Congress 1971, Information Processing 71, North-Holland Publishing Company (1972), pp 414-418

(Bochmann 1973)

Bochmann, G.V., Multiple Exits from a Loop Without the GOTO, CACM, vol. 16, no. 7, July 1973, pp 443-444

(Bohm & Jacopini 1966)

Bohm, C. & Jacopini, G., Flow Diagrams, Turing Machines and Languages with only Two Formation Rules, CACM, vol. 9, no. 5, May 1966, pp 366-371

(Brainerd 1978)

Brainerd, W., FORTRAN 77, CACM, vol. 21, no. 10, October 1978, pp 806-820

(Chapin & Denniston 1978)

Chapin, N. & Denniston, S.P., Characteristics of a Structured Program, SIGPLAN Notices, vol. 13, no. 5, May 1978, pp 36-45

(Clark 1967)

Clark, E.R., On the Automatic Simplification of Source-Language Programs, CACM, vol. 10, no. 3, March 1967, pp 160-165

(Clifton 1978)

Clifton, M.H., A Technique for Making Structured Programs More Readable, SIGPLAN Notices, vol.13, no. 4, April 1978, pp 58-63

(Cooper 1967)

Cooper, D.C., Bohm and Jacopini's Reduction of Flow Charts, CACM, vol. 10, no. 8, August 1967, pp 463,473

(Dahl & Hoare 1972)

Dahl, O-J. & Hoare, C.A.R., Hierarchical Program Structures, Structured Programming, Dahl, O-J., Dijkstra, E.W. & Hoare, C.A.R., Academic Press, 1972

(Dijkstra 1968a)

Dijkstra, E.W., Goto Statement Considered Harmful, CACM, vol. 11, no. 3, March 1968, p 147

(Dijkstra 1968b)

Dijkstra, E.W., The Structure of the "THE" Multiprogramming System, CACM, vol. 11, no. 5, May 1968, pp 341-346

(Dijkstra 1972a)

Dijkstra, E.W., Notes on Structured Programming, Structured Programming, Dahl, O-J., Dijkstra, E.W. & Hoare, C.A.R., Academic Press, 1972

(Dijkstra 1972b)

Dijkstra, E.W., The Humble Programmer, CACM, vol. 15, no. 10, October 1972, pp 859-866

(Dijkstra 1976)

Dijkstra, E.W., A Discipline of Programming, Prentice-Hall, 1976

(Ershov 1971)

Ershov, A.P., Theory of Program Schemata, The Best Computer Papers of 1971, O.R. Petrocelli (ed.), Auerbach Publishers, 1972

(Evans 1974)

Evans, R.V., Multiple Exits from a Loop Using Neither GOTO nor Labels, CACM, vol. 17, no. 11, November 1974, p 650

(Feldman & Gries 1968)

Feldman, J. & Gries, D., Translator Writing Systems, CACM, vol. 11, no. 2, February 1968, pp 77-113

(Fisher 1972)

Fisher, D. A., A Survey of Control Structures in Programming Languages, SIGPLAN Notices, vol. 7, no. 11, November 1972, pp 1-13

(Fortran 1973)

Programming Language FORTRAN, Australian Standard 1486-1973, Standards Association of Australia, 1973

(Fortran 1976)

Draft Proposed ANS FORTRAN,

SIGPLAN Notices, vol. 11, no. 3, March 1976

(Gear 1965)

Gear, C.W., High Speed Compilation of Efficient

Object Code, CACM, vol. 8, no. 8, August 1965, pp 483-488

(Gotlieb & Cornell 1967)

Gotlieb, C.C. & Cornell, D.G., Algorithms for

Finding a Fundamental Set of Cycles for an Undirected

Linear Graph, CACM, vol. 10, no. 12, December 1967,

pp 780-783

(Greibach 1975)

Greibach, S.A., Theory of Program Structures:

Schemes, Semantics, Verification, Springer-Verlag, 1975

(Gries 1971)

Gries, D., Compiler Construction For Digital

Computers, John Wiley & Sons, 1971

(Harary 1969)

Harary, F., Graph Theory, Addison-Wesley, 1969

(Hoare 1971)

Hoare, C.A.R., Proof of a Program : FIND,

CACM, vol. 14, no. 1, January 1971, pp 39-45

(Hopkins 1971)

Hopkins, M., An Optimizing Compiler Design, Proceedings of IFIP Congress 1971, Information Processing 71, North-Holland Publishing Company (1972), pp 391-396

(Hopkins 1972)

Hopkins, M.E., A Case for the GOTO, SIGPLAN Notices, vol. 7, no. 11, November 1972, pp 59-62

(Jensen & Wirth 1975)

Jensen, K. & Wirth, N. , Pascal User Manual and Report (2nd ed.), Springer-Verlag, 1975

(Knuth 1971)

Knuth, D.E., An Empirical Study of FORTRAN Programs, Software - Practice and Experience, vol. 1, no. 2, 1971, pp 105-133

(Knuth 1973)

Knuth, D.E., Fundamental Algorithms, The Art of Computer Programming, Vol. 1, 1973

(Knuth 1974a)

Knuth, D.E., Structured Programming with GOTO Statements, Computing Surveys, vol. 6, no. 4, December 1974, pp 261-301

(Knuth 1974b)

Knuth, D.E., Computer Programming as an Art, CACM, vol. 17, no. 12, December 1974, pp 667-673

(Leavenworth 1972)

Leavenworth, B.M., Programming With(out) the GOTO, SIGPLAN Notices, vol. 7, no. 11, November 1972, pp 54-58

(Ledgard & Marcotty 1975)

Ledgard, H.F. & Marcotty, M., A Genealogy of Control Structures, CACM, vol. 18, no. 11, November 1975, pp 629-639

(Ledgard, Singer & Hueras 1977)

Ledgard, H., Singer, A. & Hueras, J., A Basis for Executing Pascal Programmers, SIGPLAN Notices, vol. 12, no. 7, July 1977, pp 101-105

(Lowry & Medlock 1969)

Lowry, E.S. & Medlock, C.W., Object Code Optimization, CACM, vol. 12, no. 1, January 1969, pp 13-22

(Manna & Waldinger 1971)

Manna, Z. & Waldinger, R. J., Toward Automatic Program Synthesis, CACM, vol. 14, no. 3, March 1971, pp 151-165

(Meissner 1974)

Meissner, L. P., A Compatible "Structured" Extension to Fortran, SIGPLAN Notices, vol. 9, no. 10, October 1974, pp 29-36

(Melton 1973)

Melton, R.A., Automatically Translating FORTRAN to IFTRAN, Annual Symposium on the Interface of Computer Science and Statistics, James W. Frame (ed.), 1975, pp 291-296

(Mohilner 1978)

Mohilner, P.R., Prettyprinting Pascal Programs, SIGPLAN Notices, vol. 13, no. 7, July 1978, pp 34-40

(Morris & Wells 1972)

Morris, J.B. & Wells, M.B., The Specification of Program Flow in Madcap6, SIGPLAN Notices, vol. 7, no. 11, November 1972, pp 28-35

(Nievergelt 1965)

Nievergelt, J., On the Automatic Simplification of Computer Programs, CACM, vol. 8, no. 6, June 1965, pp 366-370

(Pascal News 14 1979)

Pascal News, Pascal Users' Group, Number 14, January 1979

(Paton 1969)

Paton, K., An Algorithm for Finding a Fundamental Set of Cycles of a Graph, CACM, vol. 12, no. 9, September 1969, pp 514-518

(Paton 1971)

Paton, K., An Algorithm for the Blocks and Cutnodes of a Graph, CACM, vol. 14, no. 7, July 1971, pp 468-475

(Pazel 1975)

Pazel, D.P., Mathematical Construct for Program Reorganization, I.B.M. Journal of Research and Development, November 1975, pp 575-581

(Peterson 1977)

Peterson, J.L., On the Formatting of Pascal Programs, SIGPLAN Notices, vol. 12, no. 12, December 1977, pp 83-86

(Peterson, Kasami & Tokura 1973)

Peterson, W.W., Kasami, T. & Tokura, N., On the Capabilities of While, Repeat and Exit Statements, CACM, vol. 16, no. 8, August 1973, pp 503-512

(Plum 1975)

Plum, T. W-S., Mathematical Overkill and the Structure Theorem, SIGPLAN Notices, vol. 10, no. 2, February 1975, pp 32-33

(Presser 1975)

Presser, L., Structured Languages, SIGPLAN Notices, vol. 10, no. 7, July 1975, pp 22-23

(Prudom & Hennell 1977)

Prudom, A. & Hennell, M.A., Some Problems Concerning the Automatic Translation of FORTRAN to ALGOL 68, SIGPLAN Notices, vol. 12, no. 6, June 1977, pp 138-143

(Raphael 1966)

Raphael, B., The Structure of Programming Languages, CACM, vol. 9, no. 2, February 1966, pp 67-71

(Rosen 1972)

Rosen, S., Programming Systems and Languages 1965-1975, CACM, vol. 15, no. 7, July 1972, pp 591-601

(Rosen, Spurgeon & Donnelly, 1967)

Rosen, S., Spurgeon, R.A. & Donnelly, J.K., Pufft - The Purdue University Fast FORTRAN Translator, Programming Systems and Languages, S. Rosen (ed.), McGraw-Hill, 1967

(Sale 1970)

Sale, A.H.J., The Classification of FORTRAN Statements, The Computer Journal, vol. 14, no. 1, January 1970, pp 10-12

(Sale 1978a)

Sale, A.H.J., Stylistics in Languages With Compound Statements, Australian Computer Journal, vol. 10, no. 2, May 1978, pp 58-59

(Sale 1978b)

Sale, A.H.J., Pascal Compatibility Report, Report No. R78-3, Department of Information Science, University of Tasmania, May 1978

(Sale 1978c)

Sale, A.H.J., Strings and the Sequence Abstraction in Pascal, Report No. R78-4, Department of Information Science, University of Tasmania, September 1978

(Sammet 1972)

Sammet, J.E., Programming Languages : History and Future, CACM, vol. 15, no. 7, July 1972, pp 601-610

(Schneck 1972)

Schneck, P.B., Automatic Recognition of Vector and Parallel Operations in a Higher Level Language, SIGPLAN Notices, vol. 7, no. 11, November 1972, pp 45-52

(Shantz et al. 1967)

Shantz, P.W., German, R.A., Mitchell, J.G., Shirley, R.S.K. & Zarnke, C.R., WATFOR - The University of Waterloo FORTRAN IV Compiler, CACM, vol. 10, no. 1, January 1967, pp 41-44

(Slater & Modell 1978)

Slater, W. & Modell, H., Structured Programming Considered Harmful, SIGPLAN Notices, vol. 13, no. 4, April 1978, pp 76-79

(Tektronix 1976)

Tektronix Inc., Plot-10 Terminal Control System User's Manual, Document No. 062-1474-00, Tektronix Inc., Beaverton, Oregon, 1976

(Tiernan 1970)

Tiernan, J.C., An Efficient Search Algorithm to Find the Elementary Circuits of a Graph, CACM, vol. 13, no. 12, December 1970, pp 722-726

(Urschler 1975)

Urschler, G., Automatic Structuring of Programs, I.B.M. Journal of Research and Development, vol. 19, March 1975, pp 181-194

(Warren 1975)

Warren, H.S. jnr., A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations, CACM, vol. 18, no. 4, April 1975, pp 218-220

(Warshall 1962)

Warshall, S., A Theorem on Boolean Matrices, JACM, vol. 9, no. 1, January 1962, pp 11-12

(Wasserman 1975)

Wasserman, A.J., Issues in Programming Language Design - an Overview, SIGPLAN Notices, vol. 10, no. 7, July 1975, pp 10-12

(Wegner 1975)

Wegner, E., Control Constructs for Programming Languages, SIGPLAN Notices, vol. 10, no. 2, February 1975, pp 34-41

(Williams & Ossher 1978)

Williams, M.H. & Ossher, H.L., Conversion of Unstructured Flow Diagrams to Structured Form, The Computer Journal, vol. 21, no. 2, May 1978, pp 161-167

(Wirth 1971a)

Wirth, N., Program Development by Stepwise Refinement, CACM, vol. 14, no. 4, April 1971, pp 221-227

(Wirth 1971b)

Wirth, N., The Design of a PASCAL Compiler, Software - Practice and Experience, vol.1, no. 4, 1971, pp 309-333

(Wirth 1973)

Wirth, N., Systematic Programming: An Introduction, Prentice-Hall, 1973

(Wirth 1974)

Wirth, N., On the Composition of Well-Structured Programs, Computing Surveys, vol. 6, no. 4, December 1974, pp 247-259

(Woodger 1971)

Woodger, M., On Semantic Levels in Programming, Proceedings of IFIP Congress 1971, Information Processing 71, North-Holland Publishing Company (1972), pp 402-407

(Woolley 1977)

Woolley, J.D., FORTRAN: A Comparison of the New Proposed Language (1976) to the Old Standard (1966), SIGPLAN Notices, vol. 12, no. 7, July 1977, pp 112-125

(Wright 1966)

Wright, D.L., A Comparison of the FORTRAN Language Implementation for Several Computers, CACM, vol. 9, no. 2, February 1966, pp 77-79

(Wulf 1971)

Wulf, W.A., Programming Without The GOTO, Proceedings of IFIP Congress 1971, Information Processing 71, North-Holland Publishing Company (1972), pp 408-413

(Wulf 1972)

Wulf, W.A., A Case Against the GOTO, SIGPLAN Notices, vol. 7, no. 11, November 1972, pp 63-69

(Yourdon 1975)

Yourdon, E., Techniques of Program Structure and Design, Prentice-Hall, 1975

(Zahn 1975)

Zahn, C.T. jnr., Structured Control in Programming Languages, SIGPLAN Notices, vol. 10, no. 7, July 1975, pp 13-15